National Library
of Canada

Acquisitions and
Bibliographic Services Branch

395 Wellington Street
Ottawa, Ontario
K1A 0N4

Bibliothèque nationale
du Canada

Direction des acquisitions et
des services bibliographiques

395, rue Wellington
Ottawa (Ontario)
K1A 0N4

Your file Votre référence

Our file Notre référence

NOTICE

The quality of this microform is heavily dependent upon the quality of the original thesis submitted for microfilming. Every effort has been made to ensure the highest quality of reproduction possible.

If pages are missing, contact the university which granted the degree.

Some pages may have indistinct print especially if the original pages were typed with a poor typewriter ribbon or if the university sent us an inferior photocopy.

Reproduction in full or in part of this microform is governed by the Canadian Copyright Act, R.S.C. 1970, c. C-30, and subsequent amendments.

AVIS

La qualité de cette microforme dépend grandement de la qualité de la thèse soumise au microfilmage. Nous avons tout fait pour assurer une qualité supérieure de reproduction.

S'il manque des pages, veuillez communiquer avec l'université qui a conféré le grade.

La qualité d'impression de certaines pages peut laisser à désirer, surtout si les pages originales ont été dactylographiées à l'aide d'un ruban usé ou si l'université nous a fait parvenir une photocopie de qualité inférieure.

La reproduction, même partielle, de cette microforme est soumise à la Loi canadienne sur le droit d'auteur, SRC 1970, c. C-30, et ses amendements subséquents.

Canada

UNIVERSITY OF ALBERTA

Graphical Interface for CASE Environment Definitions in Metaview

BY

Pius Lo ©

A thesis submitted to the Faculty of Graduate Studies and Research in partial fulfillment of the requirements for the degree of Masters of Science.

DEPARTMENT OF COMPUTING SCIENCE

Edmonton, Alberta
Fall 1995

National Library
of Canada

Acquisitions and
Bibliographic Services Branch

395 Wellington Street
Ottawa, Ontario
K1A 0N4

Bibliothèque nationale
du Canada

Direction des acquisitions et
des services bibliographiques

395, rue Wellington
Ottawa (Ontario)
K1A 0N4

Your file Votre référence

Our file Notre référence

THE AUTHOR HAS GRANTED AN IRREVOCABLE NON-EXCLUSIVE LICENCE ALLOWING THE NATIONAL LIBRARY OF CANADA TO REPRODUCE, LOAN, DISTRIBUTE OR SELL COPIES OF HIS/HER THESIS BY ANY MEANS AND IN ANY FORM OR FORMAT, MAKING THIS THESIS AVAILABLE TO INTERESTED PERSONS.

THE AUTHOR RETAINS OWNERSHIP OF THE COPYRIGHT IN HIS/HER THESIS. NEITHER THE THESIS NOR SUBSTANTIAL EXTRACTS FROM IT MAY BE PRINTED OR OTHERWISE REPRODUCED WITHOUT HIS/HER PERMISSION.

L'AUTEUR A ACCORDE UNE LICENCE IRREVOCABLE ET NON EXCLUSIVE PERMETTANT A LA BIBLIOTHEQUE NATIONALE DU CANADA DE REPRODUIRE, PRETER, DISTRIBUER OU VENDRE DES COPIES DE SA THESE DE QUELQUE MANIERE ET SOUS QUELQUE FORME QUE CE SOIT POUR METTRE DES EXEMPLAIRES DE CETTE THESE A LA DISPOSITION DES PERSONNE INTERESSEES.

L'AUTEUR CONSERVE LA PROPRIETE DU DROIT D'AUTEUR QUI PROTEGE SA THESE. NI LA THESE NI DES EXTRAITS SUBSTANTIELS DE CELLE-CI NE DOIVENT ETRE IMPRIMES OU AUTREMENT REPRODUITS SANS SON AUTORISATION.

ISBN 0-612-06503-0

Canada

UNIVERSITY OF ALBERTA

RELEASE FORM

NAME OF AUTHOR: Pius Lo

TITLE OF THESIS: Graphical Interface for CASE Environment Definitions in Metaview

DEGREE: Masters of Science

YEAR THIS DEGREE GRANTED: 1995

Permission is hereby granted to the University of Alberta Library to reproduce single copies of this thesis and to lend or sell such copies for private, scholarly or scientific research purposes only.

The author reserves all other publication and other rights in association with the copyright in the thesis, and except as hereinbefore provided neither the thesis nor any substantial portion thereof may be printed or otherwise reproduced in any material form whatever without the author's prior written permission.

(Signed) . . . .
Pius Lo
3717-103B Street
Edmonton, Alberta
Canada, T6J 2X8

Date: 22 Sept. 95.

UNIVERSITY OF ALBERTA

FACULTY OF GRADUATE STUDIES AND RESEARCH

The undersigned certify that they have read, and recommend to the Faculty of Graduate Studies and Research for acceptance, a thesis entitled **Graphical Interface for CASE Environment Definitions in Metaview** submitted by Pius Lo in partial fulfillment of the requirements for the degree of Masters of Science.

. . . .
Dr. Paul G. Sorenson (Supervisor)

. . . .
Dr. Peter Bartl (External)

. . . .
Dr. John Buchanan (Examiner)

Dr. Jia You (Chair)

Date: 20 Sept 95

To my parents,
my sister Serafina (and her new baby Justin),
and my love, Elaine

Abstract

A variety of CASE (Computer-Aided Software Engineering) methodologies and tools are presently used in development of computer software systems. A *customizable* CASE environment (also called CASE shell) is necessary to ensure effective yet flexible support of the various methods used in different development phases.

This thesis focuses on the graphical method modeling problem in Metaview, which is a CASE shell system that is capable of defining a variety of software development methods and generating automatically CASE environments from these definitions. The main goal of this research is to develop in Metaview an efficient and effective approach for defining the graphical representations of any commonly used CASE method. There are two major contributions from our research. Firstly, different approaches to graphical method definition are investigated and critiqued. Secondly, an interactive, graphical modeling tool is proposed and prototyped in order to provide intuitive and effective support for defining graphical CASE environments in Metaview.

Acknowledgements

I am very grateful to my supervisor, Prof. Paul Sorenson, for his invaluable guidance and assistance throughout my thesis research. Without his supreme supervision, this thesis could not have been accomplished. I would also like to thank specially Dr. Piotr Findeisen for his help. He spent a lot of time to guide me through the Metaview system and was always willing to explain to me everything that I was not familiar with. Thank also to the members of my examining committee, Dr. John Buchanan and Prof. Peter Bartl, for their constructive criticisms and helpful suggestions.

Special thanks are due to all members of the Software Engineering Research Laboratory. They are all very friendly people to work with. In particular, I am grateful to Narendra Ravi, Lettice Tse, and Amr Kamel for all sorts of interesting and useful discussions, whether computer related or not!

I would like to express my sincere and affectionate gratitude to my parents, Raymond and Bernadette, for their *never-ending* love and support. I would also like to thank my sister, Serafina. I always remember the time we were studying computing science together. Her help and advice greatly eased my life during the studies. Special thanks to my girlfriend, Elaine, for her all-time love, encouragement, and patience. Whenever I had problems, she was always willing to listen and encourage me. She gave me the energy (by both her support and her delicious cooking!) that made me survive my graduate studies. I am greatly indebted to her!

Finally, I would also like to acknowledge the financial support offered by the Natural Sciences and Engineering Research Council (NSERC) of Canada, and the Department of Computing Science, University of Alberta.

Contents

1 **Introduction** **1**

1.1 Motivations . . . 3

1.2 Outline of the Thesis . . . 5

2 **Background** **7**

2.1 Graphic-Oriented CASE Review . . . 7

2.2 CASE Shells and Meta-CASE Systems . . . 8

2.3 Overview of Metaview System . . . 14

2.3.1 Metaview Architecture . . . 15

2.3.2 The EARA/GE Meta-Model . . . 18

2.3.3 The Environment Constraints . . . 22

3 **Proposal of GE Language** **24**

3.1 The New Graphical Definition Language . . . 25

3.1.1 Objectives . . . 25

3.1.2 Syntax . . . 27

3.2 Examples of GE Definitions . . . 29

3.3 Conclusions . . . 33

3.3.1 Observations . . . 33

3.3.2 Recommendations . . . 34

4 **An Interactive, Graphical Modeling Tool** **36**

4.1 Objectives . . . 36

4.2 Potential Implementation Approaches 37
4.2.1 Extending OGIPS 38
4.2.2 Building a "Brand-new" Tool 39
4.3 Design of the Tool 39
4.3.1 Design Requirements 39
4.3.2 Detailed Design 44
4.4 Conclusions 52

5 The Prototype Interface — GE Definer 54
5.1 Overview of the Prototype 55
5.2 Conclusions 61
5.2.1 Observations 61
5.2.2 Summary of Results and Recommendations 65
5.3 Object Browser — The Design Proposal 68
5.3.1 Motivation 68
5.3.2 Non-Functional Requirements 69
5.3.3 Functional Requirements 70
5.3.4 Conclusions 74
5.4 Investigation on a Graphical Constraint Definer 75
5.4.1 Limitations for Defining Graphical Constraints Graphically 76
5.4.2 Conclusions 79

6 Conclusions and Future Research 81
6.1 Conclusions 82
6.1.1 Achievements and Contributions 82
6.1.2 Limitations 83
6.2 Suggestions of Future Research 83

A Summary of GE Language Syntax 95
A.1 Augmented BNF 95
A.2 Symbols Used in GE Language 97

A.3 Formal Syntax of GE Language 98

B An Example of EDL/GE Definitions — Data Flow Diagramming 103

B.1 EDL/GE Definitions of DFD 103

C System Walk-Through of GE Definer 108

C.1 A scenario of using GE Definer 108

C.2 Environment File Generated by GE Definer 117

List of Figures

1 System architecture of Metaview 16
2 Mappings between EARA and GE objects 19
3 Examples of the GE and EARA object types 21

4 The Graphical Representation of the Icon Type "process" 31
5 The Graphical Representation of the Edge Type "flow" 32

6 How the graphical modeling tool works within Metaview 45
7 Program decomposition (*highest-level*) 46
8 Decomposition of the MAIN module 47
9 Decomposition of the DIALOGS module 48
10 Decomposition of the DEFINERS module 49
11 Decomposition of the GRAPHICS module 51
12 Decomposition of the I/O module 52

13 Structure chart for GE Definer 56
14 Structure chart for "Open Environment" process 57
15 Structure chart for "Open Object Type" process 58
16 Structure chart for "Define Object Type" process 59
17 The prototype GUI layout of Object Browser 71

18 Starting GE Definer 109
19 Loading *Environment File* in GE Definer 110
20 Creating new icon type in GE Definer 111

21 Defining picture component in GE Definer 113
22 Browsing the label component in GE Definer 114
23 Warning display in GE Definer 115

List of Tables

2.1 Examples of Graphical CASE Methodologies/Tools Used in Various Software Development Phases 9
2.2 A Summary of Tools Used at the Environment Level 17

A.3 Symbols used in GE language 97

Chapter 1

Introduction

Software systems are continuously growing larger in scale and complexity due to the rapid advancement of computer technology and the increasing demands of the computer users. Since the "software crisis" in the 1970s [Pre92], production and maintenance of computer software systems have become such complex and tedious tasks that they are no longer manageable by primarily manual means. Assistance through automated software development tools or *Computer-Aided Software Engineering* (CASE) [GH89] is playing an increasingly important role in the software industry. CASE, as defined in [BCM$^+$94], "is the use of computer-based support in the software development process." In other words, it is a set of automated or semi-automated software development methods, techniques, and processes that are supported by computer programs called CASE tools.

To maximize the power and utility of these tools, a *CASE environment*, or *Software Development Environment* (SDE) [DEF$^+$87, BEM92], is desired so that a team of software developers may work on a variety of CASE tools in a common environment [FW90, Zar90]. Ideally, within such an environment, the tools are able to share project data and techniques to provide the software developers with complete support over the system development cycle. In reality, however, many existing CASE environments cannot achieve this goal because they suffer from excessive rigidity. Typically, they can only weakly support the users' commonly adopted methods and are unable to support any new, emerging methods [MRT$^+$93]. Therefore, the need for *customizable* CASE environments (also called *CASE shells*) is growing in today's software industry [LMR$^+$92].

The most important feature of these CASE shells is their capability of modeling various software development methods. These methods have to be captured by a *meta-model* in some specific representational form that is typically stored in a repository such as a special database. The modeled knowledge of these methods can therefore be used to customize the tools within the CASE shell and support the entire, or at least a portion of, the software life cycle. Not surprisingly, modeling of a software development methodology[1] is not an easy task — it requires not only the definition of the methods' concepts and rules, but also the modeling of their representational notations. The latter usually involves the use of graphics since most of the modern methods use diagrammatical techniques, such as diagrams, charts, and tables, to present their conceptual ideas and objects. Although virtually all the existing CASE shells are successful in the conceptual modeling of methods, not all seriously address issues in representational modeling.

In this thesis, we investigate the graphical definition of software development methods in Metaview[2] [Fin94d]. Our objectives are to:

1. review different approaches to graphical method modeling in Metaview;
2. propose a textual description language for defining the graphical representations of the methods, and analyze the benefits and drawbacks of this language;
3. propose and prototype an interactive, graphical tool that is used as a front-end interface for graphical modeling in Metaview;
4. examine the requirements and the limitations of the prototype tool and suggest possible future research directions.

Section 1.1 discusses the motivation for this thesis research in greater detail. Section 1.2 outlines the remaining chapters of the thesis.

---

[1] We use the term "methodology" in this thesis as "the abstract description of a class of methods" as defined by Zhuang in [Zhu94].

[2] Metaview is a research prototype CASE shell that is currently being developed by the Software Engineering Research Laboratory at the University of Alberta

1.1 Motivation

CASE tools and methodologies are used extensively in today's software industry. They are expected to be helpful in facilitating the control and coordination of the project resources, reducing substantially the costs of software development, and improving the overall quality of software products [CR88, McC89]. However, many CASE users and researchers (e.g., [JJL$^+$93, tHNV$^+$92, LST$^+$91]) point out that many of the existing CASE environments support only a limited set of methods that are not easily customizable nor extensible to satisfy the users' needs. As a result, many potential users are discouraged from introducing CASE technologies to their organizations.

In order to achieve a truly flexible, customizable and adaptable CASE environment, we need a CASE shell which is able to capture and adapt to a variety of existing software development methodologies as well as methods that emerge in the future. More importantly, a CASE shell allows customization of the captured methods to satisfy the special needs of the users. There are several research prototypes and even commercial versions of CASE shells already available in the industry — Metaview [STM88, DST89, Fin94d], MetaEdit [LST$^+$91], and Socrates [tHVW91] are some examples, and they are discussed in greater details in chapter 2 of the thesis.

To model completely any software development method, two aspects have to be considered:

- *Conceptual* knowledge

  This includes the modeling concepts provided and the constraint rules governing the use of a method [tHNV$^+$92]. The modeling concepts specify what kinds of system information can be captured and manipulated while the modeling rules specify how the modeling is to be done.

- *Representational* knowledge

  This includes the external notations used to represent the modeling concepts and the rules on how they are presented. Although the representations used by a method can be in any form such as textual reports, program codes, charts, matrices, tables, and so forth, many modern software development methodologies, especially the system specification methods, use graphics to depict the concepts of the model.

Unfortunately, many existing CASE shells focus only on the conceptual modeling and overlook the importance of the representational modeling. The modeling of the *graphical* representations is a critical and challenging problem in the *meta-CASE*[3] research area because:

1. Most of the software specification methods and many modern CASE methodologies are based heavily on graphical notations (e.g., icons, lines, and symbols) and techniques (e.g., diagrams, charts, and directed graphs) to represent their modeling concepts. Some well-known examples of these methods are Data Flow Diagraming [You89], Structure Chart [MM85], and Booch's Object-Oriented Method [Boo91]. Therefore it is important to ensure that a CASE shell is capable of modeling the graphical knowledge of any method.

2. There is no well-established principles for graphical modeling [KS94]. In most existing CASE shells, the entity-relationship (ER) model [Che76, Che83], with certain extensions, is used as the (meta-)data model for the modeling of concepts. This form of model has been shown through experience to be powerful and flexible enough to model a large variety of software development methodologies. However, no such "commonly-accepted" model has been found for graphical modeling. In many cases, the graphical representations are defined simply by "hard-coding" or using some incomprehensible script languages. Thus it is challenging to investigate an efficient and effective approach for this modeling purpose.

The current Metaview system has a meta-model called *Graphical Extension* (GE) [Fin93c], which is an extension to the conceptual *EARA* [Fin94b] meta-model, for defining graphical representations of the software development methods. The system, however, does not have any languages or interfaces based on the GE model, and therefore the graphical knowledge is *hand*-coded in the specification database. In order to improve Metaview's *methodology engineering* [HO93] capability, we need to investigate an effective approach to define the graphical part of a method. There are a number of questions that are considered in this investigation:

---

[3] Meta-CASE refers to the systems and technologies that use a *meta-system* [DST89] approach to generate customizable CASE environments. CASE shell is a kind of meta-CASE systems.

- What kinds of information do we need to capture in order to model the representational part of a method?
- What are the requirements of an efficient and effective approach to the definition of the graphic knowledge?
- What types of interfaces or tools do we need for the graphical modeling in Metaview?

These issues form the major focuses of this thesis research, and they are discussed in the following chapters.

1.2 Outline of the Thesis

The thesis is organized as follows. Chapter 2 presents an overview of the modern CASE technologies and the state-of-the-art CASE shells and their characteristics. The second half of the chapter focuses mainly on the Metaview system. The architecture, the meta-model, and the present state of the system are briefly discussed. In chapter 3, the GE language for modeling graphical elements of the methods is presented. The chapter concludes with the observations of our experience on the usability of the language, followed by a summary of its benefits and limitations. Chapter 4 presents our proposal for GE Definer — an interactive, graphical tool for the representational definition. The design of the tool and its requirements are discussed. Based on this design, a prototype tool is implemented. Key aspects of this implementation are described in chapter 5. The chapter not only presents the system structure of GE Definer and its functionality but also reports on the observations of using the tool and the limitations and problems encountered. In addition, the ideas and implementation feasibility of graphical tools for browsing a CASE environment's elements and for defining graphical constraints are introduced at the end of the chapter. Finally, the thesis concludes in chapter 6 with a summary of the major contributions of this research, followed by suggestions of future research in this area.

Chapter 2

Background

In this chapter, an overview of the current state of the CASE and Meta-CASE technologies is presented. This overview provides more motivation for the thesis research. In addition, a brief introduction to the Metaview system is presented to ensure that adequate background knowledge is provided for the discussions in the rest of the thesis. Section 2.1 presents a general review of today's *graphic-oriented* CASE technologies. Section 2.2 discusses some state-of-the-art CASE shells in the industry and their common objectives. In Section 2.3, the history and goals, the system architecture, the meta-model, and the present state of Metaview are introduced.

2.1 Graphic-Oriented CASE Review

In the 1970's and early 1980's, CASE tools and their supported methodologies were mostly text-oriented. They were primarily used in the implementation phase of the software life cycle for system coding and testing. Some examples of these tools are compilers, program editors, debuggers, and test-case generators. However, in the late 1980's, the trend in CASE technologies turned to more graphic-oriented interfaces that use pictures to describe the artifacts of a software system. This change is attributed to two main factors:

- Human Factors

Experimental findings (e.g., [PVU77]) support the claim that [McA88] "people often seem to be able to work more effectively with information when it is presented graphically than when it is in strictly textual form." Therefore graphical CASE tools can help the software developers to communicate the specifications and other concepts of the systems precisely and efficiently, and as a result, their productivity improves.

- Advanced Computer Graphics Technologies

  Thanks to the modern computer hardware technologies, virtually all computer systems can support high-quality graphical display and interactive user interfaces. Complex, yet user-friendly graphical techniques such as computer-generated diagrams, animations, multi-media and hypertext are used extensively in today's CASE tools. In addition, advanced, high-bandwidth computer networks also contribute to the success and popularity of the graphical CASE support in today's multi-user, multi-platform, and multi-site software production environments.

Graphical CASE methodologies currently support all phases of the system development cycle. Table 2.1 presents some examples of the CASE methodologies and tools used in each development phase. In particular, system analysis and design are the two phases in the software life cycle that use many different kinds of diagrammatical techniques and notations [FK92] to model the software specifications.

In summary, popularity of the graphic-oriented CASE technologies is continuously growing, and many new graphical methods are expected to emerge in the industry. Therefore a successful CASE shell must have a powerful meta-model as well as a well-defined approach for graphical modeling to support the large variety of existing and emerging software development methodologies.

2.2 CASE Shells and Meta-CASE Systems

In most organizations, a suite of CASE tools are used within a single working environment called software development environment (SDE). A SDE provides the standards and the

Software Development Phases	**Graphic-oriented CASE**
Project Planning and Management	Gantt (Timeline) Charts [Pre92], Pert Charts [Con94]
System Analysis and Design	*Structured Methodologies:* Data Flow Diagrams, Structure Charts, Decision Trees and Tables [MM85], etc. *Object-Oriented Methodologies:* OMT [R$^+$91], Booch's Method, Shlaer-Mellor's Method [SM91], etc.
Implementation	Integrated programming environments with graphical interfaces (e.g., [Orv92]), GUI toolkits [Ped92] and builders, Visual programming [Cha90]
Testing	Graphical simulations
Documentation	WYSIWYG document editors, Hypertext help-message compiler
Re-engineering / Maintenance	Rigi [MK88, MTO$^+$92], Ensemble [CTI93], REFINE [BKM90]

Table 2.1: Examples of Graphical CASE Methodologies/Tools Used in Various Software Development Phases

framework for tool integration [Was90, Zar90] in order to ensure a broader and more effective support during the system development process. More detailed discussions on various categories and models of SDEs can be found in [DEF+87, KP91].

Unfortunately, many existing SDEs suffer from the problem mentioned in the previous chapter — they are not easily customizable to the users' adopted methods nor extensible to any new methodologies. This problem limits the success of CASE in many organizations. In order to solve this problem, we need CASE shells and Meta-CASE systems.

CASE shells are customizable software engineering environments which accept the definitions of a variety of system development methods and provides flexible environments to support unlimited number of these methods. Existing CASE shells can be classified according to the degree of their flexibility and the way they (meta-)model the method specifications. For example, P. Marttiin *et al.* suggested several classification schemes in [MRT+93], and one of them is based on the customization approaches used by the CASE shells. They are categorized into three classes:

- *database oriented* — the CASE shells of this class use a higher-level language (meta-language) to define the knowledge of any CASE environment and store it in an environment specification database. The tools within the CASE shells can then be customized to a particular methodology according to its modeling concepts as well as the representational notations captured in the database.

  Examples: MetaPlex [CN89], SOCRATES [tHVW91, tHNV+92, BW], and Metaview.

- *interface oriented* — the CASE shells which adopt this approach are built with generic routines that can be easily customized to support a specific environment by associating the rules and notations used. The CASE shells in this class are supposed to work with the meta-CASE-modeling systems (and in many cases, the meta-modeling mechanisms are built-in with the CASE shells) so that the former can use the output (method knowledge) of the latter as the basis of tool customization.

  Examples: RAMATIC [BBD+89] is a representative of this class, and MetaEdit is an example of the meta-CASE systems that can be used to model method knowledge and generate output to customize the interface-oriented CASE shell like RAMATIC[1].

- *extension kit* — this approach is more limited because it cannot modify the entire environment but only extend the functionality of the existing tools.

  Example: Index Technology's Customizer^{TM} [ITCa], which is used to extend the CASE tool Excelerator^{TM} [ITCb].

In this thesis research, we focus mainly on the class of CASE shells that has built-in method-modeling support because it is a more general approach to customization of CASE environments. Throughout the rest of the thesis, we refer this class of CASE shells as *meta-CASE systems* because of their capability of meta-modeling a variety of software engineering methodologies. Some examples of the meta-CASE systems are SOCRATES, ECLIPSE, MetaEdit, and our Metaview system. In the rest of this section, the first three example systems are briefly introduced. The discussion mainly focus on their models and processes used for method modeling, and in particular, graphic knowledge modeling. In the next section, Metaview system is described in a more detailed manner.

SOCRATES

SOCRATES is a meta-CASE system developed at the Software Engineering Research Centre (SERC) in the Netherlands. The objectives of the system are to capture the knowledge of any software development method's modeling concepts (*the way of modeling*) and of its process (*the way of working*) [tHVW91]. The idea of its second objective is interesting and challenging because many other meta-CASE systems concentrate only on the modeling of a method's modeling concepts and notations. The architecture of SOCRATES is mainly divided into two levels — on the meta-level, the *meta-analyst* models the process knowledge and the product knowledge of a method into the *meta-model base* using a set of meta-model editors. On the application level, the user (e.g., system analyst) is supported by the CASE shell which is customized to his/her desired environment based on the knowledge stored in the database.

---

[1] Rossi *et al.* described how MetaEdit is used to customize RAMATIC in [RGS$^+$92].

Both conceptual and graphical definitions of a method are supported in SOCRATES. The meta-models used for the former are called Object Structures, which basically consist of three kinds of concepts — Object types, relationship types, and roles. The object types model the major concepts of a method; the relationship types define the inter-relationships between the object types; and the roles specify the participations of the object types in a relationship type. In short, this meta-model is similar to Chen's ER model [Che76]. Moreover, graphical definitions are also supported by a meta-model called Graphic Structures. There are two main constructs of the Graphical Structures: graphic object types and handle types. The graphic object types define a variety of notational conventions used in the methodologies. The handle types specify a set of reference points that are used for attachment of the graphic object instances. The Graphical Structures associated with different sets of parameters define the appearance and properties of the graphical notations. SOCRATES also has a formal graphical constraint language which specify the rules of the graphical representation of models.

In summary, SOCRATES has the well-defined, formal meta-models for both conceptual and graphical modeling. However, we feel that these meta-models are too large-grained. For example, its graphic meta-model only has the generic *graphic object type* to define all sorts of different methods' notations, comparing to *four* graphical types provided in Metaview's graphical meta-model.

ECLIPSE

ECLIPSE [BWS87] is a meta-CASE environment developed at the University of Strathclyde, Scotland, and the University of Lancaster, England. The system supports various diagraming methods that can be represented as *directed graphs*. Due to this assumption, its meta-model is fairly simple and focuses only on the methods' representational concepts. In other words, the meta-modeling language provides constructs only for graphical definitions. Since ECLIPSE assumes all CASE diagrams are represented as directed graphs, the two basic types of entity supported by the meta-model are *node* and *link*. The methods' models and rules (called *assertions*) are defined using description language. The graphical symbols are defined by a graphical tool called the SHAPES editor. All of the defined representational knowledge is then fed into a generic diagraming tool called the Design editor by which the user can draw diagrams supported by the specific method.

The major problem of ECLIPSE is its lack of a rich conceptual meta-model. In addition, its simple graphical modeling language limits its support for most modern, complex software development methods. Most current methods require tools that not only support simple drawing techniques, but can also represent the underlying modeling concepts needed to capture the specifications of a system.

MetaEdit

MetaEdit is a graphical meta-CASE environment under development in the MetaPHOR project at the University of Jyväskylä [LKK$^+$94]. Compared to SOCRATES and ECLIPSE, it is a more complete and successful meta-CASE system because:

1. MetaEdit uses a more powerful conceptual data-model called OPRR (*Object-Property-Role-Relationship*) model [Wel92]. This meta-model was more recently extended to include the concept of a graph and renamed to GOPRR[2]. This extension supports not only the graphical diagrams and symbols but also matrices and hypertext. Nevertheless, the extended model enforces the concept of representation independence such that conceptual modeling and representational modeling are separated.
2. It has a richer set of tools to support not only the application level but also the meta level. For examples, MetaEdit+ provides graphical editors — Concept editor and Symbol editor — for meta-modeling the methods' concepts and representations respectively.
3. It uses the *graphical meta-modeling techniques* to ease the definition process of the methodologies, to eliminate the time and efforts required for learning any complex meta-modeling languages, and to provide a clear visualization of the modeled method knowledge and its notations.

---

[2] Since the extension of the meta-model, the system has been changed its name to *MetaEdit+*

4. It uses not only the GUIs to display the method models on the computer screens but also a set of textual specification languages to *publish* the modeling and the representational concepts of any defined method. The output is in human readable format such that the model can be validated and reviewed by the *method engineers* and any interested readers.

5. It is able to generate report codes of method's definitions from the method specification base and use the information to *tailor* its own set of CASE tools or to customize other third-parties' CASE shells.

6. It already has a commercial version released in the market working on the PC environments.

We are most interested in MetaEdit's graphical meta-modeling techniques because it is the ultimate goal of our research in Metaview system. Since the general architecture and the approaches used by MetaEdit are quite similar to those of our Metaview system (see the next section), we have used MetaEdit system as a valuable source of reference during our research.

2.3 Overview of Metaview System

Metaview [Fin94d, STM88, DST89] is a meta-CASE system currently being developed by the Software Engineering Research Laboratory at the University of Alberta. Similar to the other meta-CASE systems', its goal is to provide the software developers with a customizable SDE support by capturing the concepts and notations of many CASE environments/methodologies in a *repository* and configuring its own set of generic tools to support the modeled methods. In other words, Metaview is also a CASE shell that provides the infrastructures and mechanisms to produce customizable CASE environment based on any defined software development methodology.

2.3.1 Metaview Architecture

The architecture of Metaview is partitioned into three levels, as shown in Figure 1. They are the *Meta Level*, the *Environment Level*, and the *User Level*.

- *Meta Level*

  The Metaview system developers define the *EARA/GE* meta-model (to be described in Section 2.3.2), the *Metaview Software Library*, the *Metaview Tools Library*, and the *Tool Components Library*. The Metaview's Software Library is different from the Tools Library as the former consists of tools for the system users[3] (e.g., environment editors, compilers, and database management utility software) while the latter contains the generic, customizable tools for the CASE users. The Tool Components Library is a collection of reusable object class definitions and modules. They are used for construction of new tools and communications among them. In summary, this Meta Level is the fundamental platform of the whole Metaview system.

- *Environment Level*

  The SDEs and their supported methodologies are defined at this level with the assistance of the software built at the Meta Level. Two major activities at the Environment Level are *method modeling* and *system configuring*. The method modeling task in Metaview is accomplished by a set of tools and modeling languages that model the four aspects of any software development method (SDM) based on the EARA/GE data-model. These four parts of an SDM are the *conceptual model*, the *conceptual constraints*, the *graphical representations*, and the *graphical constraints*. They are discussed in greater details in the following sections. In particular, the GE language and the graphical tool, GE Definer, proposed in this thesis are tools designed for the method modeling process. They are used as an interface between the method definer and the Metaview system to capture effectively the method's graphic knowledge. The system configuring process is done by setting up the Metaview's *Database Engine* to

---

[3] The system users, in this context, are the Metaview developers and method engineers, who use the Metaview software to build an customized SDE.

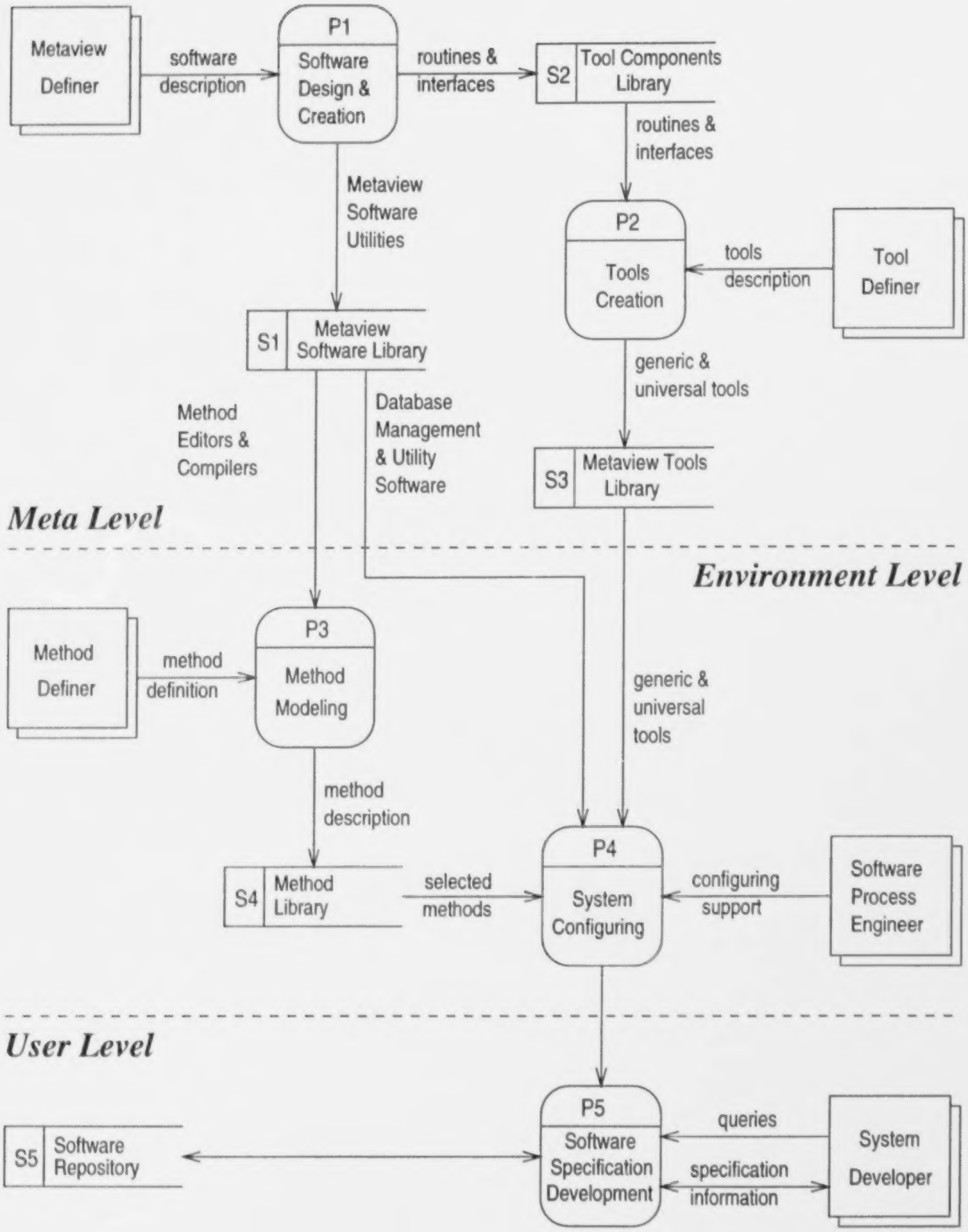

Metaview Definer
software description
P1
Software Design & Creation
routines & interfaces
S2 Tool Components Library
routines & interfaces
Metaview Software Utilities
P2
Tools Creation
tools description
Tool Definer
S1 Metaview Software Library
generic & universal tools
Method Editors & Compilers
Database Management & Utility Software
S3 Metaview Tools Library
Meta Level
Environment Level
Method Definer
method definition
P3
Method Modeling
generic & universal tools
method description
P4
System Configuring
S4 Method Library
selected methods
configuring support
Software Process Engineer
User Level
P5
Software Specification Development
S5 Software Repository
queries
specification information
System Developer

Figure 1: System architecture of Metaview

support a particular SDE based on the knowledge of the modeled SDM and that of the software process. Table 2.2 summarizes the tools that have been built and being used at the present version of Metaview.

Types of Tools	**Available Tools**	**References**
Definition Languages	EDL/GE ECL	[GLM94] and also see Chapter 3 [Fin94c, Fin94f]
Method Definers/Editors	OGIPS GE Definer	[Mac91] see Chapter 5
Compilers	EDL Compiler	[GM93]
SDM Repositories	Database Engine	[Fin93b]

Table 2.2: A Summary of Tools Used at the Environment Level

- *User Level*

  The end-users of Metaview, such as system analysts and developers, are supported by the configured SDEs generated at the Environment Level for use at this level. The SDEs consist of the generic CASE tools and the knowledge of the specific SDMs defined in the upper levels. At the present stage, Metaview only has one tool available for this User Level which is called MGED (*Metaview Graphical Editor*) [Fin93d]. It is a graphical editor for software specifications that supports any modeled SDM through the communications with the Metaview's *Project Daemon* [Fin94e] and stores the specification information in the database (also called *software repository*) for report and reuse purposes. The Project Daemon is a server program that provides the Metaview tools with a uniform interface to the Database Engine for concurrent access of system information. It is expected that in future a variety of tools will be implemented for this level so that the Metaview's end-users will benefit from a better CASE support over their software development process.

For more detailed description on the Metaview architecture, the reader may refer to [Fin94d].

2.3.2 The EARA/GE Meta-Model

The fundamental platform of Metaview is based on the meta-modeling data-model called EARA/GE (*Entity-Aggregate-Relationship-Attribute with Graphical Extension*) originally proposed by McAllister in his PhD. thesis [McA88]. Since then the model has been refined and extended to satisfy the requirements of the Metaview system and the whole variety of SDMs. The current versions of EARA and GE are described in [Fin94b] and [Fin93c] respectively. In this section, we briefly introduce the EARA model and then concentrate on the discussion of the GE graphic model. The interested reader should refer to the above-mentioned references for more details of these meta-models.

The EARA meta-model is based on the Entity-Relationship data-model [Che76] but is extended with features such as *aggregation* and *classification*. The major elements of EARA are *Entity* type, *Relationship* type, *Aggregate* type, and their associated properties (*Attributes*).

Entities are real-world (conceptual) objects of a system that can be modeled and specified by an SDM. Some examples of the typical entities of a software system are modules, data, variables, events, states, and so forth.

Relationships are the associations between entities and aggregations. In any system, each object has relationship(s) with one another, and information is transferred through these *links*. For instance, a module (entity) of a system invoking another can be represented as a relationship between them. Each relationship has a number of roles which determine the functions played by the *participants* of that relationship.

Aggregates are *composite* objects that represent a *heterogeneous* collection of entities and relationships. Aggregation is a special form of relationship necessary for modeling today's SDMs which must deal with complex system objects that can be viewed as subsystems. Therefore, the addition of the aggregation feature significantly increases the modeling power of the original ER model in SDM modeling.

Most existing SDMs have object types for modeling similar but different *classes* of system objects. Thus during the method modeling, it is convenient to define a common class (supertype) for the similar object types (subtypes) such that common attributes of the supertype can be *inherited* to its subtypes. Both EARA and GE models supports such specializa-

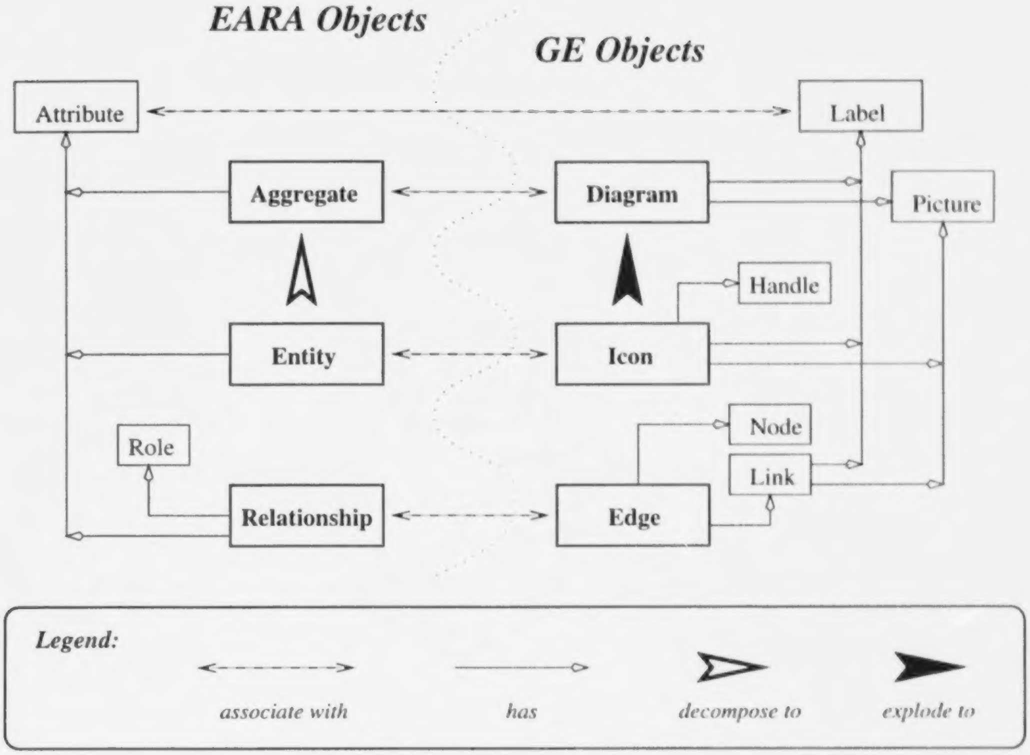

EARA Objects
GE Objects
Attribute
Label
Aggregate
Diagram
Picture
Handle
Entity
Icon
Role
Node
Link
Relationship
Edge
Legend:
associate with
has
decompose to
explode to

Figure 2: Mappings between EARA and GE objects

tion hierarchies for object types by classification.

The GE model is an extension to the EARA model — while the EARA model defines the conceptual elements of an SDM, the GE model specifies their representational counterparts. The current version of GE model supports only two-dimensional, non-animated graphics. It is because most existing diagrammatical methods originate from "pencil-and-paper" methods and thus rarely use three-dimensional or animated graphics. The connections between the conceptual and the graphical definitions are achieved by introducing three graphical object types — *Icon* types, *Edge* types, and *Diagram* types — which define the graphical representations of their corresponding Entity, Relationship, and Aggregate types, respectively. Figure 2 illustrates the EARA and GE elements and the mappings between them.

Icons represent graphically entities of a system. They are displayed on computer screen as rectangular areas, filled with *pictures* and *labels*. Pictures represent the graphical symbols used by an object type in a specific method, and labels represent the text that displays the values of the associated attributes. Each icon is an independent graphical objects that can be created and manipulated by the user. In addition, each icon that participates in an edge (relationship) should have a special defined component called a *handle*. Handles are sets of regions (though they may be as small as a point) on an icon where a particular edge can touch. The definition of the handles is important because many SDMs, for example SADT [MM87], attach meanings to different positions of the *join points* of an icon. Not every point of the icon can be used for any edge. In addition, as an EARA entity can *decompose* to an aggregate, an icon can also *explode* to a diagram. This feature is necessary because many modern SDMs support system modeling in different levels such that an object in one level may represent a subsystem in a lower level. Therefore, if an entity is represented by a *child aggregate*, its corresponding icon can explode to a diagram that represents the aggregate.

Edges represent relationships that have entities as participants. They are displayed as the edge patterns with a set of *links* (represented by lines) and *nodes* (represented by points) similar to those of undirected graphs. The nodes represent the roles of the relationship types and form the connection points of the links. The links represent a *visual* connections between nodes and may associate with arbitrary number of pictures and labels.

Finally, a diagram represents an aggregate. Each diagram is composed of icons and edges that represents a system or subsystem description. A diagram may also have any number of pictures and labels. Diagrams are displayed on a computer screen as a window that can be manipulated by the user.

Figure 3 illustrates some sample instances of these GE object types and their associated EARA types using a DFD example.

In addition to these three GE object types, a *picture type* is used to define the geometrical pattern that is the pictorial basis for an icon, edge, or diagram. Any picture pattern used in the graphical object types is constructed by a selection of six pre-defined picture primitives: *point*, *line*, *arc*, *circle*, *box*, and *text*. For most existing SDMs, this set of primitives

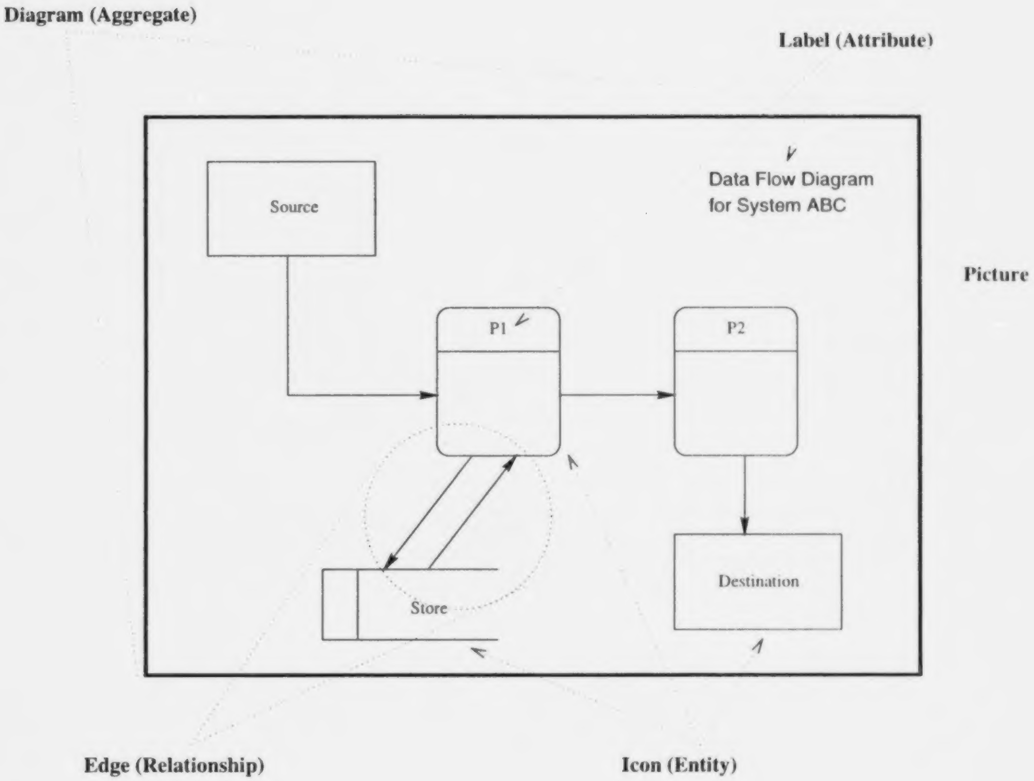


Diagram (Aggregate)
Label (Attribute)
Data Flow Diagram
for System ABC
Source
Picture
P1
P2
Store
Destination
Edge (Relationship)
Icon (Entity)

Figure 3: Examples of the GE and EARA object types

is adequate to build the graphical notations required.

Furthermore, virtually every graphical element type used in GE has its own set of properties defined. These graphical properties determine the representational characteristics, such as dimensions, color, and other visual effects, of that element. In addition, the model allows any number of new properties to be added to any GE elements and thus increases the flexibility and extensibility of the graphical support in Metaview.

2.3.3 The Environment Constraints

Constraints of both the conceptual and the graphical definitions of a method ensure the correct use of the model and the appropriate diagrammatical layout of its representations. These constraints are described as a set of predicates defined by the method engineer according to the characteristics of the modeled SDM. They are stored in the Database Engine with the method's specifications.

The conceptual constraints provide two kinds of checking: *completeness* and *consistency*. Completeness checking reports the missing parts of any partially defined model, and consistency checking ensures a consistent model — that is, objects within the model have to be consistent with one another and satisfy all pre-defined conditions. In a similar manner, graphical constraints guard the graphical consistency of the objects. They are mainly the diagraming rules applied by the SDM to ensure the resulting model representations look "as expected". For many SDMs, constraints are added to define a set of guidelines for producing more readable and better structured diagrams.

Presently, Metaview has two different versions of ECL (*Environment Constraint Language*) [McA88, Gad93] for defining conceptual constraints. A new version has been designed [Fin94f, Fin94c] and is being implemented which supports graphical constraints. Furthermore, in this thesis, we also investigate the feasibility of implementing a graphical tool for graphical constraints definition.

Finally, further details on SDE modeling in Metaview can be found in the references [Fin93a] and [Fin94a]. The former presents some environment modeling guidelines, and the latter describes an example of the complete definition of Data Flow Diagram environment using Metaview.

Chapter 3

Proposal of GE Language

Method modeling is performed in Metaview at the *Environment Level* as described in Section 2.3.1. To model completely a method, the method definer has to define the *four* main parts of the method. They are the conceptual model and constraints, and the graphical representations and constraints. Support for the definition of these "method parts" is developed at the *Meta Level* and provided to the method environment definer. For example, the EARA meta-model and its graphical extension (GE), as described in Section 2.3.2, were developed to capture the knowledge of the method model concepts and their graphical representations. However, in the previous version of Metaview, there was very limited support for the method modeling process — only two description languages, EDL and ECL, were developed for defining the conceptual environments and constraints. In particular, there was no tool support for the graphical modeling. The graphical definitions of a method must be hand-coded directly in the Metaview's method specification database [Fin93b]. To solve this problem, this thesis research first started with a proposal of a new description language, called "**GE language**", to define the graphical representations[1] used in a method.

Section 3.1 discusses the objectives and requirements of the proposed GE language. Section 3.2 presents some examples of the GE definitions to illustrates how the GE language models various representational notations of a method. Section 3.3 summarizes the benefits and limitations of the GE language based on our experience with the language, and several recommendations are presented.

---

[1] As mentioned in Section 2.3.3, the graphical constraints are to be defined by a new version of ECL [Fin94c] which is currently being implemented.

3.1 The New Graphical Definition Language

Our motivation of designing a new graphical definition language was to:

- produce a *simple* solution to the problem of limited graphical method modeling support in Metaview;
- make observations on the results of this solution and examine the potential requirements and problems of the graphical modeling task;

The preliminary design idea of the language originated from a graphical extension of the original EDL language proposed by McAllister in his thesis [McA88]. However, we made substantial changes and improvements over his proposed language when we designed the new GE language. Because Metaview's graphical extension model has been modified and improved since McAllister's work, the language McAllister proposed is no longer compatible with the current Metaview system. Furthermore, the syntax of his language was not as declarative and readable as we desired. For example, the representation parts of a graphical type are defined using "geometric procedures" which are like the lower-level function calls used in common procedural programming languages.

The following section discusses the objectives of the GE language. Section 3.1.2 briefly describes the syntax of the language.

3.1.1 Objectives

To provide adequate support for the definition of graphical method elements, there are five major objectives that the GE language must fulfill:

- *Readable Definitions*

  The GE language must be easy to understand because the method definer must use it to *write* the definitions of a graphical method environment without assistance of any tool support. Moreover, the language should use high-level language constructs, for example "English-like" keywords and statements, to ensure that the resulting definitions are easily readable. The readability of the GE definitions is important because these definitions may be published and reviewed by many interested parties, including people who may not be familiar with GE language and Metaview.

- *Complete Definition Support*

  The language should provide complete graphical definition capability such that it can define every graphical element supported by the GE meta-model.

- *Graphical Inheritance Hierarchies*

  Because many diagraming methods have some graphical object types that are similar to one another, it is more efficient to "factor out" the common components and properties of these graphical types and build a *supertype* object to hold the common definitions. Other object types can therefore defined as *subtypes* of this supertype object and inherit its definitions. GE language should support such inheritance hierarchies.

- *Consistent with EDL*

  The design of GE language should be consistent with the EDL language. For example, the GE language should use the same, or at least similar, syntactic rules and conventions as EDL. Because the method definer has to use both languages to define a complete (i.e., both conceptual and graphical) method model, it is easier to learn and use the new GE language if it is consistent with EDL. Therefore, GE language should be designed as an extension to the EDL language.

- *Independent Graphical Definitions*

  Although both GE and EDL languages should be similar in their syntax and structures, it is important to enforce the independence of their definitions. In other words, the graphical definitions described in GE language should be "self-contained". This approach has an advantage of making the definitions more flexible because the method definer would be allowed to modify the graphical definitions of a method without affecting the conceptual counterparts. This can help the definer concentrate on the graphical modeling of a method without worrying about what is defined in the conceptual part. Another advantage is that the independence of graphical object definition allows *separate* inheritance hierarchies for graphical elements and thus enhances modularity of the GE definitions.

However, on the other hand, a graphical definition can *never* be completely separated from its conceptual counterpart because every graphical object type must represent a corresponding conceptual types in a method environment. Therefore, the GE language should allow *mappings* between conceptual and graphical objects, while preserving the independence of their definitions.

3.1.2 Syntax

In order to fulfill the objectives described in the previous section, the syntax and constructs of GE language were carefully designed. First of all, the syntax of the language is simple. For example, the definition of a graphical object type is declared as a single "type" *statement*. A statement is composed of a number of *clauses* that define the components of that object type. A clause can be further broken down into several *subclauses* that define various information and properties of a graphical component. In short, the syntax of the GE language is well-structured because the definition of a graphical type is built up by pieces of information in different levels of details.

Moreover, each GE statement is constructed by a sequence of "*keyword–information*" pairs. That is, each piece of information defined is preceded by a pre-defined keyword to identify its meaning. For example, to define the radius of a circle primitive, a string "`RADIUS` r" (r is the radius value) is used. The keywords used in GE (and EDL too) are all meaningful *English* words or phrases. Hence, due to the simple, structured syntax as well as the meaningful keywords, the definitions described in GE language are readable and easily understandable.

To maintain the consistency with EDL, GE language uses the same syntactic rules and conventions as EDL's. For examples, both languages are case-sensitive and they both use a *semicolon* ";" to terminate a type statement. In addition, they also share some commonly used keywords.

The GE language supports mapping between a graphical object type and a conceptual counterpart by means of type naming, that is, the definitions of both types are linked by using the same, unique type name.

The GE language completely supports all the graphical types and elements defined in the GE model. The language consists of:

- *Four kinds of graphical type statements* — three of them define the graphical object types: `DIAGRAM_TYPE`, `ICON_TYPE`, and `EDGE_TYPE`. The fourth one is the `PICTURE_TYPE` statement which defines a geometrical pattern using the selection of picture primitives.
- *Six kinds of picture primitive clauses* — these clauses are `POINT`, `LINE`, `ARC`, `CIRCLE`, `BOX`, and `TEXT`. They are used only in the `PICTURE_TYPE` statements to construct picture patterns.
- *Five kinds of graphical component clauses* — these clauses define the graphical components attached to a graphical object type. These components are `PICTURES`, `LABELS`, `HANDLES`, `NODES`, and `LINKS`.

In addition, since the EDL contains language constructs to support *subtyping*, the GE language uses the same constructs to build the inheritance hierarchies of graphical types. For examples, the keyword "`GENERIC`" is used to declare an object type as a supertype, and "`IS_A`" to specify a *parent* by which the type is inherited.

In conclusion, the syntax and other constructs of the GE language are designed to fulfill the expected objectives. The formal syntax of GE language is summarized in Appendix A. For more detailed information about the syntax of EDL and GE languages, the interested reader should refer to [GLM94]. Some examples of GE definitions will be described in the next section.

3.2 Examples of GE Definitions

The examples in this section are cited from the definitions of the Data Flow Diagraming (DFD) method. The complete EDL and GE definitions of this method are presented in Appendix B. A more detailed description on how the complete DFD method (including the constraints) is defined using EDL/GE and ECL can be found in [Fin94a].

To demonstrate all kinds of statements and clauses in GE language, three different kinds of object types in the DFD method are selected:

Diagram Type — "`any_level`"

Any diagram used in the DFD method is represented by a diagram type `any_level`. This diagram type has a corresponding *aggregate* type defined in EDL with the same type name. A definition of this diagram type in the GE language might be:

```
DIAGRAM_TYPE any_level
  PROPERTIES (x_size = 595, y_size = 770);
```

The first line specifies the kind of this GE type and the type name. The second line is a *clause* that defines some graphical properties of this diagram type. In this example, the two properties defined are the horizontal and vertical dimensions of the diagram. This `DIAGRAM_TYPE` definition does not contain other clauses because the diagrams used in DFD are represented by plain, rectangular windows on the screen without any pictures or labels.

Icon Type — "`process`"

The `process` icon type graphically represents the `process` entity type defined in the conceptual definitions of the method. The GE definitions of this icon type are:

```
CONSTANT FontH  =  13;    /* the default font height */
CONSTANT ProcW  =  80;    /* process icon width */
CONSTANT ProcH  =  8*FontH+2;    /* process icon height */
CONSTANT ProcR  =  18;    /* the radius of the corner rounding
                             for the process picture */

PICTURE_TYPE process_pic
  BOX FROM (0, 0) TO (ProcW, ProcH) RADIUS ProcR
  LINE FROM (0, 2*FontH+1) TO (ProcW, 2*FontH+1);

ICON_TYPE process
  LABELS (id_number AT (ProcW/2, FontH+1)
             PROPERTIES (x_size = 55, y_size = 2*FontH),
          name AT (ProcW/2, 4*FontH+2)
             PROPERTIES (x_size = ProcW-1, y_size = 4*FontH),
          form AT (ProcW/2, 7*FontH+2)
             PROPERTIES (x_size = ProcW-1, y_size = 2*FontH))
  PICTURES (process_pic)
  PROPERTIES (x_size = ProcW+1, y_size = ProcH+1)
  HANDLES (flow.*
          AT ((ProcR .. ProcW-ProcR, ProcR .. ProcH-ProcR)));
```

The definitions consist of an `ICON_TYPE` statement, a `PICTURE_TYPE` statement, and a few `CONSTANT` statements which define the symbolic constants used in the other two statements. Figure 4 shows the graphical structure of this defined icon type.

The `PICTURE_TYPE` statement defines the picture pattern as shown in the figure. It consists of two picture primitives — a box defined with the coordinates of its two diagonal corners and the radius of the rounded-corners, and a line defined by the coordinates of its two ends.

The `ICON_TYPE` statement defines the icon's graphical properties (i.e., its dimensions) and its associated components — labels, pictures, and handles. The `LABELS` clause defines three labels to be displayed at the specified coordinates to show the values of the attributes `id_number`, `name`, and `form`, which are defined in the corresponding entity type. Each of these labels has its own properties defined in a `PROPERTIES` subclause.

The `PICTURES` clause specifies that the picture pattern `process_pic` , defined in the `PICTURE_TYPE` statement, is used in this icon type. The location and orientation of a picture pattern can also be specified, optionally, in this clause. But in this example, they are

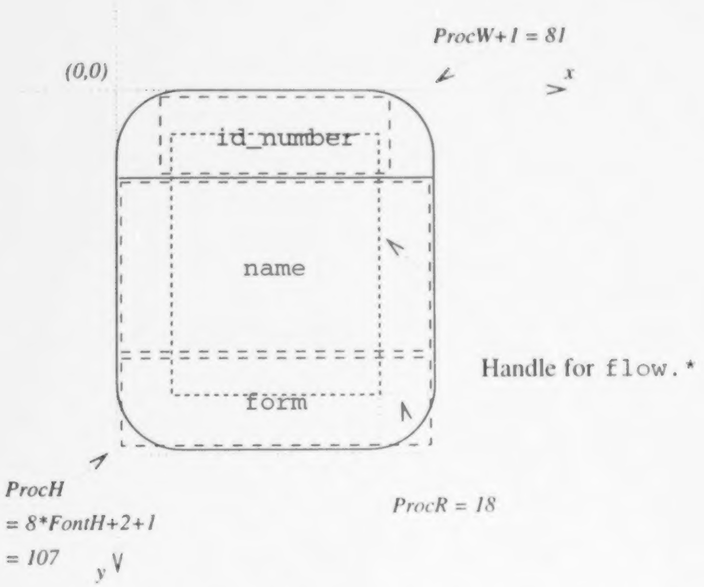

ProcW+1 = 81
(0,0)
x
id_number
name
Handle for flow.*
form
ProcH
= 8*FontH+2+1
= 107
ProcR = 18
y

Figure 4: The Graphical Representation of the Icon Type "process"

not specified because the picture pattern is placed at the default location (i.e., the origin of the icon type) and in default orientation (i.e., the picture pattern is not *rotated*) within the icon.

Finally, the `HANDLES` clause defines a rectangular region within this icon where any role ("*" in GE language means *all*) of the edge type `flow` can touch.

Edge type — "`flow`"

The edge type `flow` graphically represents its corresponding relationship type. Figure 5 shows this edge pattern. The definition of this edge type consists of a `PICTURE_TYPE` statement and an `EDGE_TYPE` statement:

```
PICTURE_TYPE arrowhead
  LINE FROM (0, 0) TO (-15, -5)
  LINE FROM (0, 0) TO (-15, 5);
```

```
EDGE_TYPE flow
  NODES (source      AT (0, 0),
         data        AT (100, 20),
         destination AT (200, 0))
  LINKS (FROM source TO destination
            LABELS (frequency AT (100, -20))
            PICTURES (arrowhead AT destination));
```

The picture pattern defined in the `PICTURE_TYPE` statement is a simple "arrowhead" pattern that is used by the link of the edge. The edge type has three nodes that are defined in the `NODES` clause. The nodes represent the roles defined in the `flow` relationship type, and each of this node has a specific location[2].

The `LINKS` clause defines a visible *line pattern* that connects the two nodes[3] `source` and `destination`. The *arrowhead* picture and a label are also attached to the link, as defined in its subclauses.

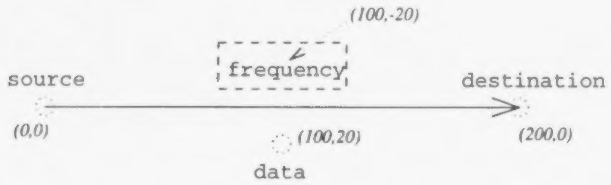


Figure 5: The Graphical Representation of the Edge Type "flow"

---

[2] It is important to note that the locations defined in an edge type are not *absolute* locations. When the modeled method is finally used to model a system specification at the *User Level* in Metaview, the edge pattern can be scaled. The locations specified merely "provides some guidelines as to how a 'good looking' edge instance can be formed" [Fin93c].

[3] It is important to note that the locations used in a link can be specified using the node names rather than the coordinates. This feature can ensure that the link always connects with the nodes, even when they are moved.

3.3 Conclusions

In this research, a graphical definition language — the GE Language, was designed to fulfill the proposed objectives. The language has been used to define successfully various SDMs in Metaview; for example, the Data Flow Diagraming method, Ward and Mellor's Methodology for real-time systems development [GFS$^+$94], and Zhuang's Object-Oriented Modeling [Zhu94]. Some observations of the benefits and limitations of the GE language are discussed in the next section. Section 3.3.2 presents our recommendations which are based on the observed results from the use of the graphical definition language.

3.3.1 Observations

From our experience on the use of GE language, the observed benefits and limitations to graphical method modeling in Metaview are summarized as follows:

Benefits

- *A Simple Approach to Graphical Modeling*

  The GE language provides a simple approach to the definition of graphical method environments. The language was designed as an extension to EDL and its syntactic structures and conventions are simple, consistent and well-structured. It is easy for the method definer to learn and use the new language.

- *Support of All Graphical Definitions*

  The GE language supports completely the GE meta-model. The language contains various kinds of statements and clauses that can be used to define all kinds of graphical object types and their associated components and properties. Thus the GE language is a sufficient language for the method definer to understand and use in order to model the graphical representations of a method.

- *Publication of the Graphical Definitions*

  Since the GE language is a textual, declarative language, its definitions can be published for verification and review purposes. Furthermore, the language's simple syntax and meaningful keywords ensure the readability and understandability of the GE definitions. Therefore, interested parties, even those who are not familiar with the GE language, can study and understand the graphical definitions of a method.

Limitations

- *Tedious Definition of Graphical Elements*

  A major problem is that the GE language is not able to provide the user with visualization of the defined graphical elements and properties. For example, our experience indicates that the definition of picture patterns using GE language is awkward. The method definer is required to specify the coordinates and other information (e.g., radius) of every picture primitive used in a picture pattern. There is no way to examine the "look" of the defined picture until the definition is compiled and later used in the graphical editor MGED. In short, based on our experience, the use of a textual language is a tedious way of defining the graphical representations a method.

- *Lack of Interactive Support*

  The only existing tool support for GE language is the EDL/GE compiler, and there is no support available during the process of graphical modeling. Our observations indicate that novice users find difficulty in learning and using the GE language because there is no interactive assistance, such as a help facility and real-time error checking, for the tedious modeling task.

3.3.2 Recommendations

Based on the observed limitations of the GE language, we realized the need for interactive support in defining the graphical aspects of a method environment. An interactive, graphical tool that makes the graphical modeling task easier and more *intuitive* is required and is proposed in the next chapter. By "intuitive", we mean that the proposed graphical tool should reduce the tedium of the graphical modeling process. Moreover, the tool should provide the method definer with interactive modeling support such as immediate visual feedback in a graphical form, real-time validations, on-line assistance, and dialogs for user inputs.

On the other hand, GE language still has some unique benefits that are important to the Metaview system. For example, it makes publication of the graphical definitions of a method possible. We recommend that the GE language should be used in Metaview as an *intermediate* representation for a method's graphical specifications. This representation form should be accessible by the existing (and future) tools and used for transferring graphical method information between these tools.

Chapter 4

An Interactive, Graphical Modeling Tool

As indicated from our observations in using the GE language, we need an interactive, graphical tool to provide a more efficient and effective way for graphical modeling in Metaview. In this chapter, we discuss various design issues of such a modeling tool. Section 4.1 presents the objectives of the tool. Section 4.2 discusses two possible approaches that were considered in the implementation of the tool and justifies our final decision — building the tool from scratch. Section 4.3 presents the design proposal for the tool. Finally, Section 4.4 presents the conclusion of this chapter.

4.1 Objectives

The main motivation for the proposal of the graphical modeling tool is to provide the method definer with an intuitive and effective mechanism to define the graphical representations of a software development method. The objectives of this tool are summarized as follows:

- *An efficient and effective approach to graphical method modeling*

  The proposed tool eases the task of graphical modeling because it is more intuitive to define graphical objects *graphically* as opposed to using a textual description. For example, elements such as picture patterns, and properties such as picture placement and colors, are easiest to define and display graphically on the computer screen because the user is provided with instant visual feedback on what s/he defines. The visual feedback is extremely helpful in minimizing errors and reducing the time wasted in "trials and errors" analysis during the method modeling process.

Moreover, the graphical tool also allows real-time interactions with the user. The use of interactive dialogs, menus, and toolboxes ensures the user inputs to be handled promptly and the visual feedback to be provided immediately. The interactive approach to method modeling provides the method definer with a better user support such as on-line, context-sensitive help facility and real-time validation for defined method environment.

In summary, the "what you see is what you get" working environment offered by the tool not only improves the overall productivity of the method definer but also ensures the quality of the products (i.e., the method's graphical definitions).

- *A graphical browsing facility for methods' representational definitions*

  The tool works not only as an *editor* but also as a *browser* for the graphical elements of a method. It supports both creation of new graphical object types as well as access to the existing ones. The method definer can use the graphical modeling tool to review the graphical object types of a previously defined method environment.

- *Generation of compatible and reusable methods' graphical specifications*

  The tool produces the graphical definitions of a method that are usable by the Metaview system and understandable by the method engineers. Therefore, the tool-generated specifications of the graphical elements must be compatible with the current configuration of Metaview and usable by the other software within the system. On the other hand, the specifications must also be presented in human readable format so that they can be published for verification and review.

4.2 Potential Implementation Approaches

Based on a clear identification of the major objectives of the proposed tool, we considered two general approaches to the system development — *extending an existing tool called*

OGIPS or *building a new tool from scratch*. Section 4.2.1 discusses the limitations of the first approach, and Section 4.2.2 explains why we selected the second approach.

4.2.1 Extending OGIPS

One possible approach was to extend an existing graphical tool developed for Metaview called OGIPS. OGIPS (*Object-oriented Generator for Interactive Picture Specifications*) [Mac91] was designed and built by MacKenzie at the University of Saskatchewan. It is a graphical tool for defining the picture patterns used by the graphical notations in a method. Since the tool is basically a simple picture drawing program, it must be extended to support the definition of other graphical elements and properties in order to fulfill our objectives. However, we recognized a number of limitations on the usability and extendibility of OGIPS. The major problems that make this extension approach unfavorable are summarized as follows:

1. *OGIPS' user interface is not easily extendible.* Since the tool was originally designed for defining only picture patterns, all the command buttons on its toolbar are used for picture editing functions, and more critically, the interface does not make use of menus. Therefore we found it difficult to modify the tool's interface, without re-writing most of its GUI codes, to adapt to a new set of graphical modeling functions.

2. *OGIPS was improperly designed and has conflicts with its own operational platform — X Window System [Ped92].* Moreover, OGIPS was implemented using only the X library routines instead of any GUI toolkits such as Motif [You90]. As a result, its interface cannot be easily enhanced. Thus fixing all these problems requires an extensive re-design of the program.

3. *OGIPS' output is no longer compatible with the current version of Metaview.* The original goal of OGIPS was to generate binary codes that, after being compiled and executed, displayed the defined pictures on the computer screen. Unfortunately, such binary programs are no longer useful because the way that Metaview stores and displays graphical symbols was changed.

4.2.2 Building a "Brand-new" Tool

Another approach to the development of the proposed graphical tool is to start from scratch. Although this approach requires a lot of time and effort, it is a more efficient and effective solution than extending OGIPS. In particular, the objectives of a tool that models *all* graphical elements of a method are quite different from those of a picture drawing tool like OGIPS. It is evident from the unsatisfaction characteristics of OGIPS cited in the previous section, extending OGIPS would require a complete re-design of the program. Therefore, we decided to prototype a "brand-new" tool that satisfy all of these design requirements for an effective graphical extension definition tool.

4.3 Design of the Tool

In this section, we present the design issues of the proposed graphical modeling tool. Section 4.3.1 discusses the design requirements of the tool according to the objectives described in Section 4.1. The architecture of the tool and the detailed design of each module are discussed in Section 4.3.2. Furthermore, the section also explains how the proposed tool works within the Metaview system environment.

4.3.1 Design Requirements

To ensure that the proposed tool can achieve the general objectives described in Section 4.1, we constructed a list of requirements to be followed throughout the design and the implementation phases. These requirements are grouped according to the major objectives of the tool and are described in the rest of this section.

User-friendliness

A good design of the tool's user interface is important because the main goal of the proposed tool is to provide a more intuitive way to define the graphical representations of a method. A well-designed GUI can certainly ease such a tedious modeling task. To produce a tool that is user-friendly yet powerful enough for graphical modeling, we need to make use of modern GUI technologies such as WIMP[1].

The GUI is expected to be implemented on a window environment (e.g., X Window System) such that multiple windows can be used to present different types of model information simultaneously. For example, while the main application window is used to edit a graphical object type, a separate window can be used as a *toolbox* that contains the buttons and controls for various editing functions. The concept of such "toolboxes" is important in our design because they can be used for defining different kinds of GE object types. Each toolbox is a "floating" dialog window on the computer screen, and the user can use the "tools" provided to define every element of a particular graphical type. An advantage of using the toolbox is that it provides the user with a complete view of what parts and properties of an object type can be defined. This feature is particularly helpful for novice users who are not sure what GE type components are definable. Moreover, other GUI features like pull-down menus, status bars, pop-up dialogs, and "point-and-click" mouse actions all improve the user-friendliness of the tool.

Another design issue that affects the usability of the tool is how much control the tool imposes on the method definer. In particular, it is important to decide whether the tool should allow the user to have the freedom to work with either a restricted or arbitrary number of graphical objects. In our prototype development it was decided that the tool should allow the user to work with a single graphical object type at a time (though each object type may have many associated components and properties). Therefore, if the user wants to edit an object type other than the current one, s/he has to save the current object type, quit the current "definer toolbox", and select the toolbox for the other object type. This approach removes the confusion caused by dealing with more than one GE type at a time and guides the user to complete the definition process. The latter benefit is desirable because modeling a complete graphical object type often requires definition of many sub-components, and it is not unusual for even an experienced method definer to miss some of them. On the other hand, a potential limitation of this approach is that it reduces the tool's flexibility which may be preferable to some *expert* users. However, so far our experience on method modeling in Metaview indicates that even a method modeling expert normally works with one object type at a time. Hence this "single-object working environment" approach was viewed as more user-friendly than the alternative approach of dealing with multiple objects.

---

[1] This acronym stands for Windows, Icons, Menus, and Pointing devices.

Real-time Interactions

In order to provide the user with the benefits of real-time interactions, the tool should offer:

- *Immediate Visual feedback*

  Whenever the user defines or modifies a graphical element, the tool should update that object definition and provide immediate visual feedback. Such feedback includes re-displaying the changed graphical object and updating the corresponding information shown on the toolbox and the main status bar. This allows the user to *see* the effects of what s/he did and spot any mistakes immediately.

- *Real-time error checking*

  Sometimes the user may not notice a mistake s/he makes by simply looking at the screen. In those cases, it is important for the tool to detect these errors and warn the method definer. Therefore, whenever the user makes any changes, the tool should validate those changes to make sure they are valid and consistent. In case of problems, warning messages can be displayed to the user by pop-up dialogs. This feature not only ensures the correctness of the definitions but also makes the modeling process more efficient.

- *Interactive dialogs*

  Different kinds of dialogs are used to help the user interact with the tool. Inputs from the user and output from the tool itself can be handled through these dialogs to provide real-time feedback to the user. For example, in many cases, the user can input textual data, such as the name of a graphical object type, through a data entry dialog. The dialog provides a selection of valid entries for that data item permitting the user to choose from a list instead of typing in the data manually. This approach can reduce significantly the chances of making errors such as mistypes and invalid data. On the other hand, the tool can also communicate with the user using the dialogs in real-time. For example, when the user performs an invalid action, the tool warns the user and suggests an alternative action through a pop-up message dialog.

- *On-line help*

  The tool keeps track of the current state of the definition process and provides on-line, context-sensitive help dialogs upon the user's request. The help messages explain specific features of the tool and give useful hints to guide the user through the modeling process. We expect that, in future, the help facility can be improved to a *tutorial* style that offers "step-by-step" assistance for the user to complete the definition of a graphical object type. An example of such a facility in a commercial product is the "wizard" feature in the latest version of Microsoft Excel^{TM}[2].

Information Browsing and Presentation

Our proposed tool is used not only as a definer assistant for methods' representations but also as a browser for the defined graphical types. No such graphical browsing tool exists in the present version of Metaview. This feature supports the improvement and customization of pre-defined methods in Metaview by assisting the user in *reviewing* the graphical definitions of method environments.

To present the definitions of the graphical types in browsing mode, we make use of a canvas and a toolbox. The canvas is the main drawing area that shows the appearance of the graphical elements and their placement within the object type. The toolbox, on the other hand, displays other information about the object type. For example, it can show the object type's name, its supertype's name, and its properties' values. Furthermore, the toolbox can inform the user of which and how many the object type components are defined and of which are yet undefined.

To allow loading of the pre-defined graphical representations, the tool must have access to their definitions. Since most of the existing graphical methods in Metaview are defined in GE language, our proposed tool must be able to read the GE definitions as input and display the graphical types on the screen.

---

[2] Microsoft Excel^{TM} is the product of Microsoft Corporation.

Compatibility with Metaview

To ensure compatibility with the current Metaview system, the design of the tool is based on the GE meta-model and the GE language. Everything that can be defined using the proposed tool is supported by the GE model and can be alternatively defined in the GE language. This requirement guarantees that our tool can always support existing GE definitions, and always generate GE definitions for its defined graphical representations. The GE language is the universal graphic definition language that is expected to be supported by all, present and future, Metaview tools.

While the GE language is used as an external representation for the graphical notations of a method, we also need an internal counterpart that can be used by the tool to store temporarily the graphic definitions. This common, internal data structure, called the Metaview Symbol Table [Fin95], consists of classes of data objects and associated functions to store and manipulate the definitions of the graphical object types.

In addition, our proposed tool also has the same "look and feel" as the other existing Metaview tool — MGED. Both have similar Motif-style GUIs [OSF92]. Future graphical tools of Metaview should also have their user interfaces built according to the same style.

Finally, we must also be concerned about the extendibility of the tool. In future, the tool will be:

- improved to provide more advanced graphical modeling/editing features to support more complex graphical representations used in emerging methods.

- modified to adapt to possible changes or enhancements to the current GE model that improve its modeling power. For example, the capability of defining new graphical types and properties may be required.

Hence we need to consider carefully the tool's architectural and implementational design to make sure it is easily modifyable. The design process begins by using classical functional decomposition to break down the tool's structure into many small and maintainable modules. Each of these modules are responsible for a particular task. For instance, the GE component definer module of the tool is divided further into a number of sub-modules that define components, such as label, picture, handle, etc. This modular design not only makes the program easier to maintain but also provides more flexibility for adding new features in future.

4.3.2 Detailed Design

We briefly introduce how the proposed graphical modeling tool works within the Metaview system environment and then discuss the program architecture and some detailed design issues of each module.

Working within Metaview Environment

The proposed tool is to be used by the *method definer* for the *method modeling* process at the *Environment Level* as described in Section 2.3.1. Figure 6 depicts how this tool works with the method definer and the Metaview system.

The current design of our tool does not include details of the *Picture-Type Definer* module. It is because the module is basically a simple picture drawing program, with many of capabilities available in OGIPS, as previously described in Section 4.2.1. In this thesis, our goal is to focus on a prototype graphical tool that defines the GE object types with their various components and properties. Because the output of OGIPS is not usable for current Metaview system and OGIPS has some inherent design problems with its a Picture-Type Definer module must be added to a future version of our tool. At present, the picture types (which represent the geometrical patterns used in the GE types) are hand-coded in the GE language.

The method definer models new graphical types or edits existing ones using the tool based on her/his knowledge of a method's graphical model. The modeling task is done

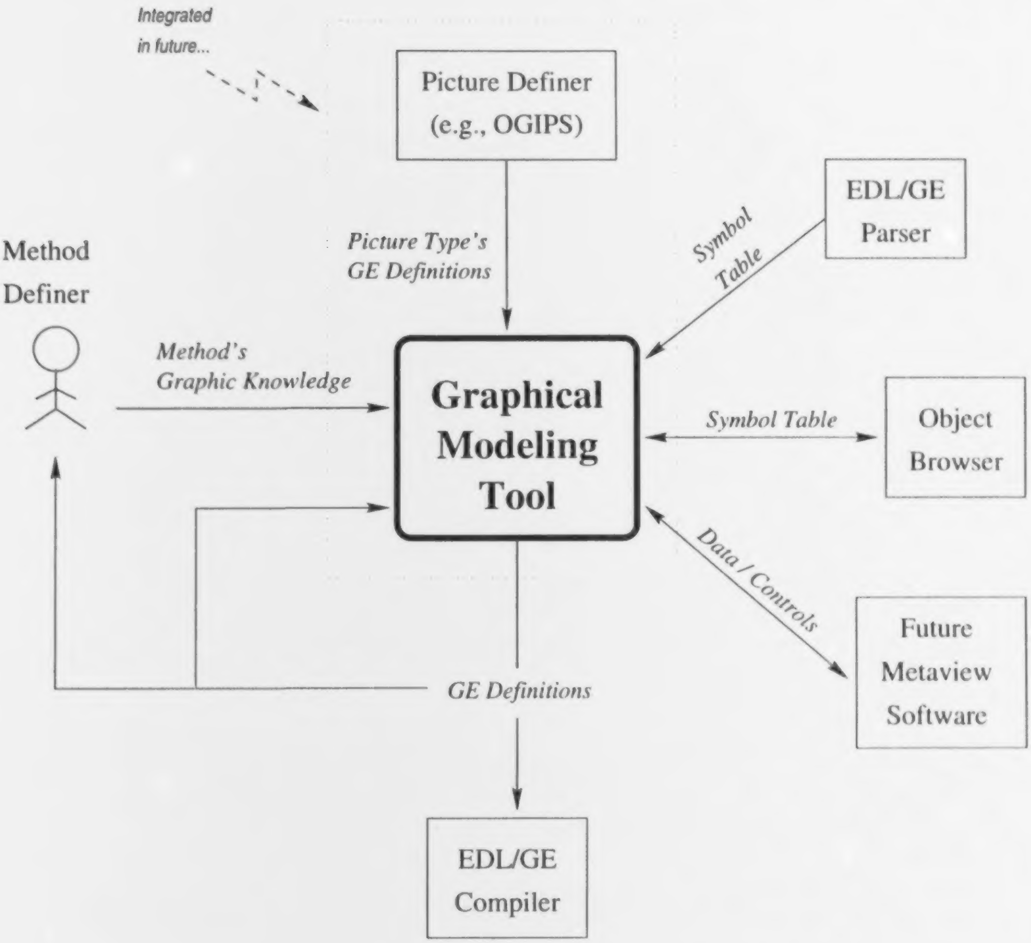

Integrated
in future...
Picture Definer
(e.g., OGIPS)
EDL/GE
Parser
Symbol
Table
Picture Type's
GE Definitions
Method
Definer
Method's
Graphic Knowledge
Graphical
Modeling
Tool
Symbol Table
Object
Browser
Data / Controls
Future
Metaview
Software
GE Definitions
EDL/GE
Compiler

Figure 6: How the graphical modeling tool works within Metaview

graphically through interactions between the user and the tool. In order to "load" a predefined method environment, the tool invokes the *EDL/GE Parser*[3], which reads the definitions of the selected environment and returns the information through the Metaview Symbol Table. With this approach, the modeling tool will also work with other Metaview tools in the future. One of these tools is the *Object Browser*; the design of which will be discussed in Section 5.3.

When the graphic definitions are completed and validated, the tool generates the output specifications in GE language. The GE definitions are then reviewed by the method definer and other interested readers, compiled by the *EDL/GE Compiler* and stored in the database engine, or reused by the tool itself for future modification.

Program Architecture and Design

The architecture of the tool consists of five main modules as shown by the decomposition diagram [MM85] in Figure 7. The functionality and the detailed design of the main modules are introduced in the rest of this section. In addition, for each module, a decomposition diagram is presented to illustrate the module's internal structure.

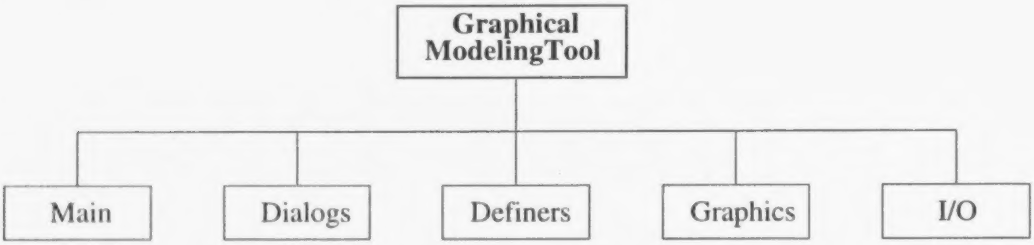


Figure 7: Program decomposition (*highest-level*)

- *MAIN* (see Figure 8)

  This module consists of the main program and other library routines used throughout the program. There are generally two types of libraries defined in the tool — *core library* and *GUI library*. The core library consists of the commonly used functions and procedures that can be shared by the different modules of the tool. The GUI library contains the common subroutines that are useful for creating and configuring the interface components. These subroutines mainly make use of the X and Motif library functions to manipulate the GUI widgets and events [You90].

---

[3] The EDL/GE Parser is currently being developed by Dr. P. Findeisen as part of the EDL/GE Compiler implementation project.

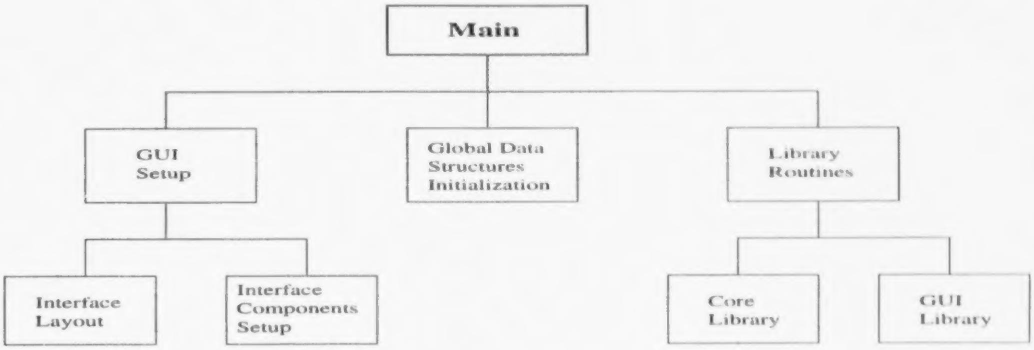


Figure 8: Decomposition of the MAIN module

Moreover, the MAIN module is also responsible for setting up the tool's main interface and initializing the global data structures. The main interface is basically a window that consists of a menu bar (for executing commands), a status bar (for showing information of the current graphical type), and a canvas area (for displaying and manipulating the graphical elements). The global data structures consist of a number of variables used by the tool and two types of data structures. The first one stores the information about the current environment such as its filename and a reference to the environment object's definition in the Symbol Table. The other data structure holds the information of the current graphical type such as its name, *kind* (Diagram, Icon, or Edge), filename, and a reference to the GE object type's definition in the Symbol Table.

- *DIALOGS* (see Figure 9)

  This module consists of sub-modules that handle different types of dialogs that constitute the major communication facilities between the user and the tool. The types of dialogs supported are:

  File I/O — contains the file selection dialogs created using the Motif's *convenience functions* [You90]. The dialog shows the directory of the file system and helps the user to select the desired file.

  Selection — consists of various kinds of data selection dialogs through which the user can input a data item by selecting a value from a list of choices.

  Warning — is a generic message dialog that is used to display different kinds of warning messages generated by the tool. The dialog uses different icons and heading titles to represent various kinds of messages such as warnings, errors, advice, etc.

  Help — is a message dialog that can display pre-defined help messages based on the current context of the tool.

  Miscellaneous — contains all the other special-purpose dialogs for specific interactions between the user and the tool.

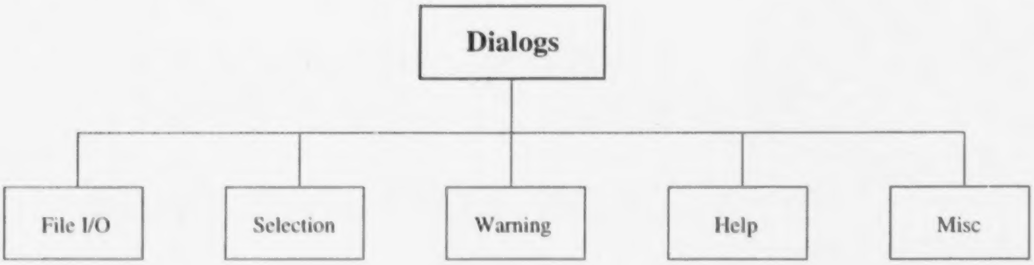


Figure 9: Decomposition of the DIALOGS module

- *DEFINERS* (see Figure 10)

  This module manages the "toolboxes" for defining the graphical elements and properties ,and therefore forms the core part of the tool. The definer toolboxes are divided into two general groups. The first group is the *GE Types Definer* which consists of toolboxes to define the GE object types (Diagram, Icon, and Edge Types) and their representational properties. The other group is the *GE Components Definer* which defines various types of GE components such as pictures, labels, and handles, etc.

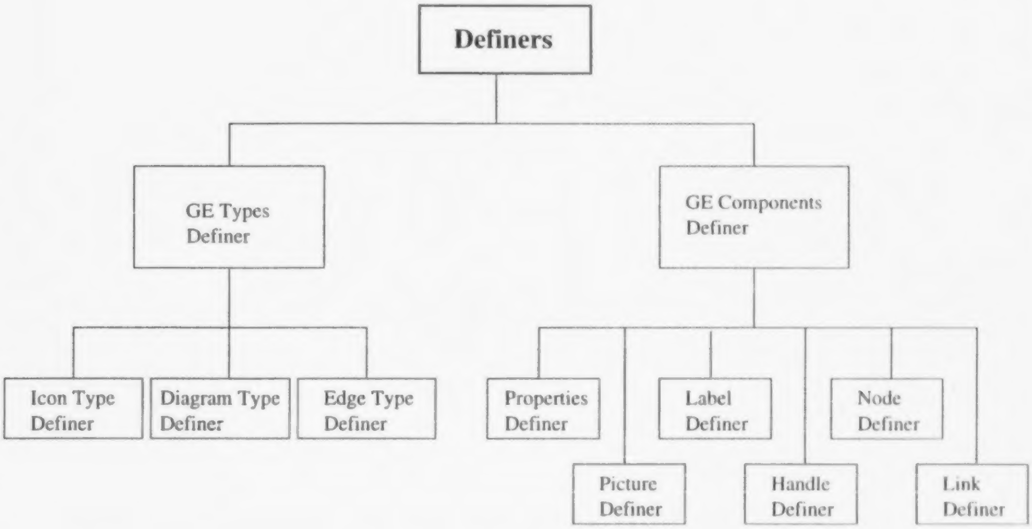

Definers
GE Types Definer
GE Components Definer
Icon Type Definer
Diagram Type Definer
Edge Type Definer
Properties Definer
Label Definer
Node Definer
Picture Definer
Handle Definer
Link Definer

Figure 10: Decomposition of the DEFINERS module

Each of these toolboxes is implemented as a "floating" dialog that can be moved by the user around the screen at his/her convenience. The dialog contains a group of controls ("tools"), which usually include push buttons, toggle buttons, text-entry boxes, sliders, etc., for defining the properties of a particular GE element.

In addition to its defining capability, the toolbox is also used to browse through a list of graphical types[4]. This is necessary because sometimes the user needs to select one of the existing GE types for modification, and there must be some way for the user to search through the selection and pick the choice. Hence the toolbox is designed to support such a browsing/selecting facility. Each definer module is able to retrieve a group of GE elements and display the information on each element on the toolbox. The user can choose whether to select the current type or examine the next type by pressing the toolbox's buttons. In the *browsing mode*, all the controls on the toolbox are disabled so that the user can never accidentally change the definitions of the types. Once the user selects a type, the toolbox changes back to the *defining mode* to permit the definer to modify the chosen GE type.

---

[4] In this context, a graphical type is referred to any type of GE elements — it may be a GE object type of an environment or a GE component type of an object type.

- *GRAPHICS* (see Figure 11)

  This module contains sub-modules that manage the main canvas, display the graphical objects on screen, and perform graphic editing functions. The *Canvas Manager* is responsible for setting up the canvas and handling its events. Moreover, it also supports *scrolling* of the canvas. This feature is needed because it is unreasonable to limit the sizes of the objects being defined by the physical size of the window. Therefore the canvas has two scrollbars which control the horizontal and vertical scrollings respectively. To prevent loss of orientation, the x and y axes (represented as two straight lines) are always updated and refreshed whenever the canvas is scrolled.

  Moreover, the GRAPHICS module also has the *Display Module* that takes care of all the display functions for the GE elements. The module can display any type of GE element by traversing its data structure and showing every defined graphical components within the type. The picture patterns are handled by the *Pictures Display* submodule. It invokes some general presentation routines for doing picture rotation and coordinates mapping, and Motif's convenience functions for displaying the geometrical primitives on screen.

  Finally, the *Graphic Editing Module* supports various graphic manipulation techniques such as rubber-banding and drag-and-drop methods. More advanced graphic techniques can be added in future to support more complex editing tasks.

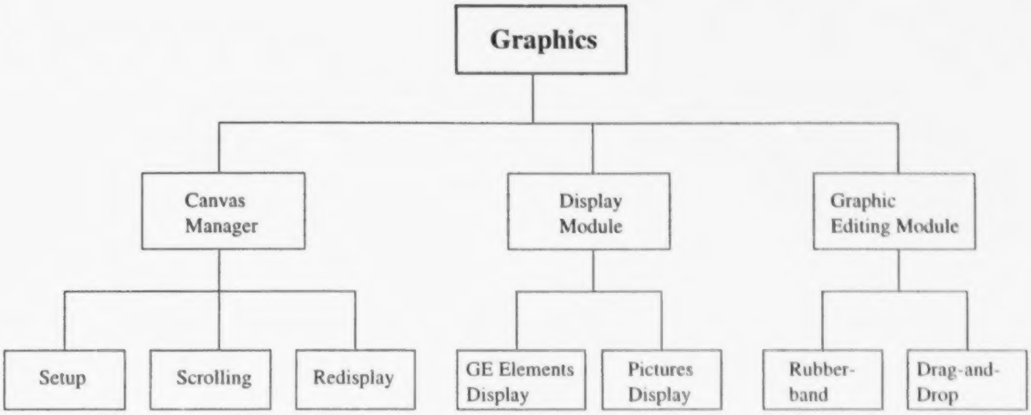

Graphics
Canvas Manager
Display Module
Graphic Editing Module
Setup
Scrolling
Redisplay
GE Elements Display
Pictures Display
Rubber-band
Drag-and-Drop

Figure 11: Decomposition of the GRAPHICS module

- *I/O* (see Figure 12)

  This module is responsible for handling all inputs and outputs of the tool. In particular, there are two types of I/O — *Environment I/O* and *Object File I/O*. The Environment I/O uses the definitions in GE language to represent a group of *completely-defined* graphical object types. In other words, these GE definitions form the specifications of a method's graphical environment. The Environment I/O is supported by two sub-modules: GE Parser and GE Generator. The GE Parser is developed as part of the EDL/GE Compiler and is not covered in the scope of our discussion here[5] On the other hand, the GE Generator is implemented as a system of sub-modules such that each generates GE definitions of a particular kind of GE types or components.

  The Object File I/O module maintains a special file that stores temporarily the specifications of object types. These object types may be partially defined and need future modification. The reason why a special file format is needed is that the EDL/GE Parser does not accept *incomplete* or *inconsistent* GE definitions. Therefore it was

---

[5] For more information on the GE Parser and the EDL/GE Compiler, the reader may contact Dr. Piotr Findeisen (Department of Computing Science, University of Alberta) who is responsible for the project.

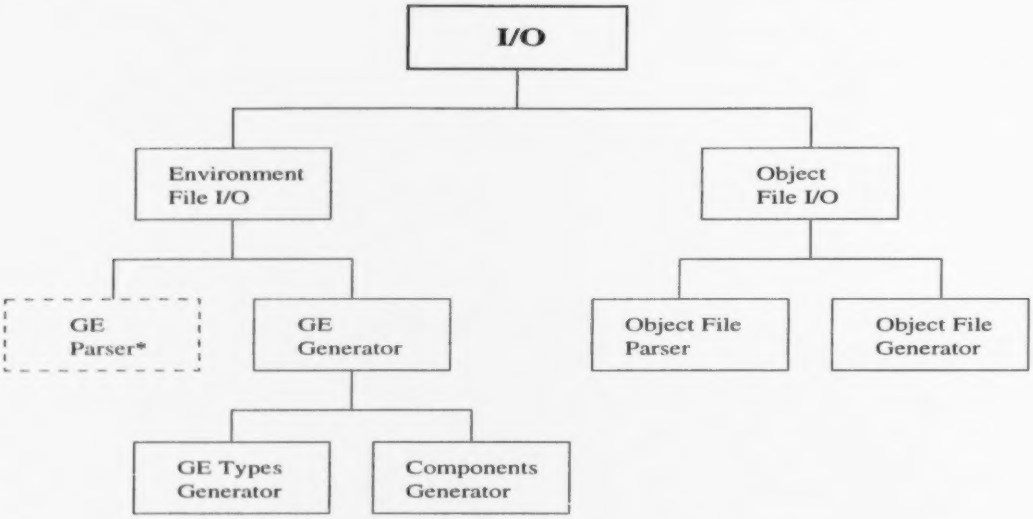

I/O
Environment File I/O
Object File I/O
GE Parser*
GE Generator
Object File Parser
Object File Generator
GE Types Generator
Components Generator

Figure 12: Decomposition of the I/O module

necessary to design a new file format, and implement both a generator to save the specifications of these temporary types and a parser to read them. These two modules are again composed of many smaller sub-modules that support specifications of each kind of GE elements.

4.4 Conclusions

In summary, to make graphical method modeling in Metaview more efficient and effective, a graphical tool for defining the method's GE object types is definitely needed. The major objectives of such graphical modeling tool are identified and summarized as follows:

- The graphical modeling tool provides the method definer with an effective and intuitive support for defining interactively the graphical representations of any method;
- The tool supports both definition as well as browsing of graphical elements in a method;

- The tool generates graphical definitions that are usable by the Metaview system and understandable by the users.

Based on these objectives, we produced the design requirements for such a graphical modeling tool. Furthermore, the detailed design of the proposed tool was also produced and used in the implementation of a prototype called "*GE Definer*". This prototype version of GE Definer successfully fulfills most of the design requirements described in this chapter. In Chapter 5, we will discuss this prototype graphical modeling tool regarding to its implementation and functional issues.

Chapter 5

The Prototype Interface — GE Definer

In this chapter, we describe the prototype version of our proposed graphical modeling tool. At the present stage, although more enhancements and extensions are still expected, the prototype successfully fulfills most of the objectives and requirements discussed in the previous chapter. The current version of the tool is called "**GE Definer**" because its main purpose is to *define* the *GE* object types of any SDM's graphical representations. To ensure our design matches the product requirements, we produced quickly a prototype interface using Visual BASIC^{TM}[1], before spending significant effort on the actual implementation. This early prototype demonstrated our design ideas for the proposed tool, and the functionality and layout of its user interface. We used our observations on this prototype to refine the original design and finally implement the tool.

Section 5.1 presents an overview of GE Definer that covers several implementational issues and a general discussion on how the tool works. Section 5.2 concludes the discussion of GE Definer with a summary of the observations obtained during the implementation and the use of the prototype. It also discusses the contributions and limitations of GE Definer with respect to graphical method modeling in Metaview. Finally, some future extensions to GE Definer and research directions for better graphical modeling support are suggested. In Section 5.3, we presents the proposal of "**Object Browser**". We discuss the motivation of designing this browser, the tool's design and implementation requirements, and its contributions to the method modeling process in Metaview. Finally, Section 5.4 examines the implementation feasibility of a graphical tool for defining graphical constraints. Initially, we expected that the graphical constraints were like the graphical representations that could be defined intuitively and effectively by a graphical tool. From our investigation, however, we realize that most graphical constraints are too complex to be expressed clearly and precisely using a graphical language.

---

[1] Visual BASIC [Orv92] is an application development environment produced by Microsoft Corporation. It runs under Microsoft Windows^{TM}.

5.1 Overview of the Prototype

GE Definer is a prototype version of our proposed graphical modeling tool, which runs on Sun workstations under UNIX and X Window System. The tool's functional core is implemented using C++ programming language [Str87] and its GUI part uses OSF/Motif toolkit. It also makes use of some library modules of Metaview such as the EDL/GE Parser and the Symbol Table.

The program structure of GE Definer is composed of a number of modules. Each of these modules is invoked to perform a specific function with respect to the graphical modeling process. The system description of GE Definer is depicted by the structure chart[2] shown in Figure 13. In the rest of this section, a general description on how GE Definer operates is presented.

To start defining the graphical representations of an SDM, an *environment* needs to be opened. An environment is a collection of all the conceptual and graphical definitions of the object types used in an SDM. Opening an environment in GE Definer requires either creating a new environment object or loading a pre-defined one from an *Environment File*. The Environment File consists of the method specifications in EDL/GE language. Figure 14 shows the procedures required for opening an environment.

We recommend that, before using GE Definer, the method definer should first prepare an

---

[2] The structure charts presented in this section are produced using MGED, the graphical editor for Metaview, which is customized to support the structure chart diagraming method.

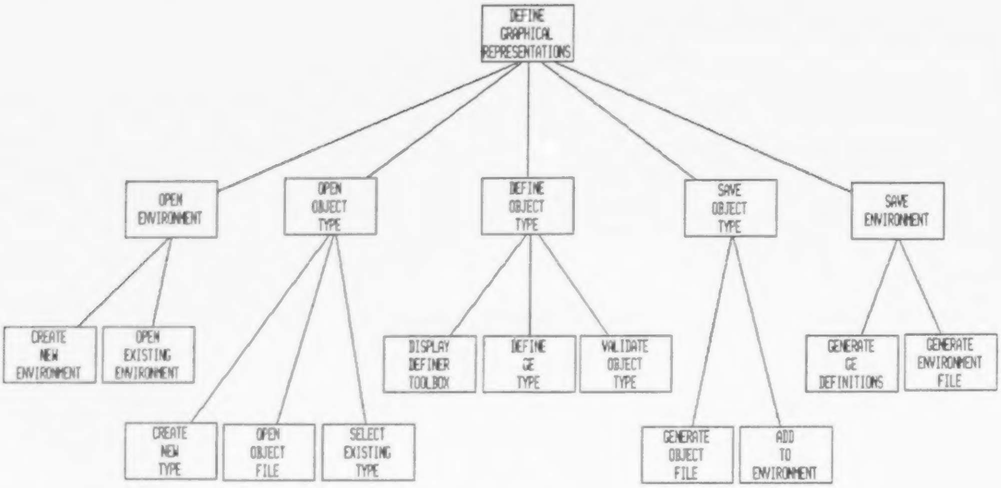

DEFINE GRAPHICAL REPRESENTATIONS
OPEN ENVIRONMENT
OPEN OBJECT TYPE
DEFINE OBJECT TYPE
SAVE OBJECT TYPE
SAVE ENVIRONMENT
CREATE NEW ENVIRONMENT
OPEN EXISTING ENVIRONMENT
DISPLAY DEFINER TOOLBOX
DEFINE GE TYPE
VALIDATE OBJECT TYPE
GENERATE GE DEFINITIONS
GENERATE ENVIRONMENT FILE
CREATE NEW TYPE
OPEN OBJECT FILE
SELECT EXISTING TYPE
GENERATE OBJECT FILE
ADD TO ENVIRONMENT

Figure 13: Structure chart for GE Definer

environment with the conceptual definitions and the specifications of the picture patterns. The environment is loaded to GE Definer and parsed by the EDL/GE Parser module. The method definer can therefore use the existing conceptual types and picture patterns to model the graphical representations. The user can also choose to start with a "brand-new" environment. GE Definer generates a new environment by creating a new object of the environment class and initializes it with a title and description given by the user.

Once an environment is opened, it becomes the *current* environment, and GE Definer displays its information on the status bar. At this stage, the user can open a new or existing GE object type for definition or modification.

A GE object type can be opened in three ways. Figure 15 shows the structure chart for this process. The user may choose to create and define a new graphical object type by specifying the kind of GE type s/he wants. GE Definer then creates a new object type[3] of the specified kind and assign to it a set of standard properties initialized with their default val-

---

[3] In this section, the terms "object type" and "GE type" are used interchangeably. They both mean a graphical object type of a modeled method.

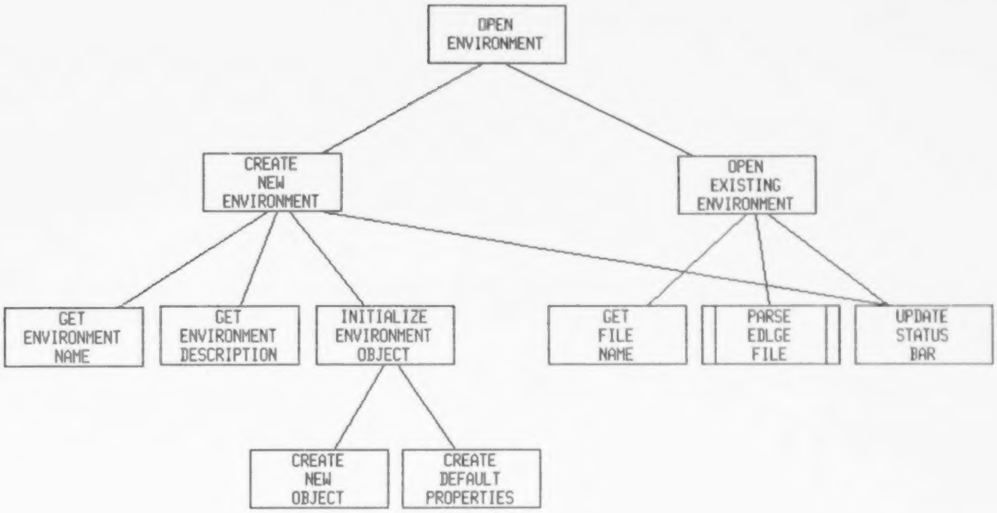

OPEN ENVIRONMENT
CREATE NEW ENVIRONMENT
OPEN EXISTING ENVIRONMENT
GET ENVIRONMENT NAME
GET ENVIRONMENT DESCRIPTION
INITIALIZE ENVIRONMENT OBJECT
GET FILE NAME
PARSE EDLGE FILE
UPDATE STATUS BAR
CREATE NEW OBJECT
CREATE DEFAULT PROPERTIES

Figure 14: Structure chart for "Open Environment" process

ues.

If the user wants to modify a pre-defined object type, s/he needs to either load an *Object File* or select a GE type from the current environment, depending on where the desired object type exists. An Object File, as described in Chapter 4, is a special file used internally by GE Definer to store the specifications of a GE object type. To open an object file, the user specifies its filename using a file selection dialog. The parser module then reads the file and stores the object type's specifications into the GE type object definition.

If the desired GE type already exists in the current environment, GE Definer traverses the environment's data structure and produces a list of valid choices so that the user can select the desired object type from the list.

Once a GE type object is opened, it becomes the *current* object type and is ready to be defined or modified.

Defining a GE type involves three major processes as shown in Figure 16. First of all, GE Definer displays a "definer" toolbox. The toolbox is constructed according to the kind

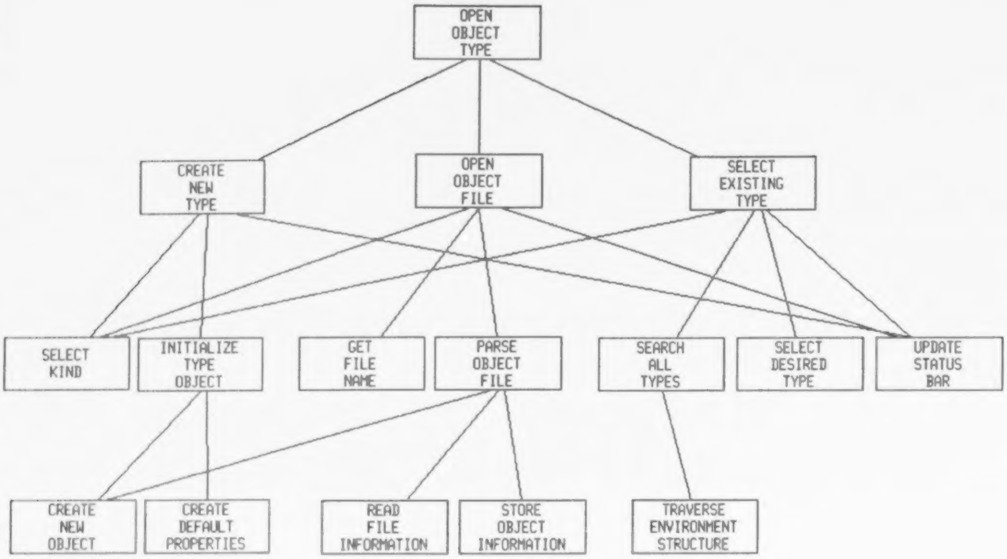

OPEN OBJECT TYPE
CREATE NEW TYPE
OPEN OBJECT FILE
SELECT EXISTING TYPE
SELECT KIND
INITIALIZE TYPE OBJECT
GET FILE NAME
PARSE OBJECT FILE
SEARCH ALL TYPES
SELECT DESIRED TYPE
UPDATE STATUS BAR
CREATE NEW OBJECT
CREATE DEFAULT PROPERTIES
READ FILE INFORMATION
STORE OBJECT INFORMATION
TRAVERSE ENVIRONMENT STRUCTURE

Figure 15: Structure chart for "Open Object Type" process

of the current object type. It consists of controls ("tools") for defining the components and properties of the GE type. Furthermore, it also displays the information of the current object type. Once the toolbox is built and displayed on the screen, GE Definer invokes the graphics display module to display the object type. This module traverses the entire structure of the GE type and calls various sub-modules to draw the "appearance" of the defined graphical components (such as pictures and labels) and displays the associated graphical properties (such as object sizes).

Once the "definer" is set up, the user can define two aspects of the current GE type:

- *Properties*

  The graphical properties of the GE type represent the rules and attributes that control the appearance of the types' instances, which are eventually used in the actual system modeling. For example, a pair of common graphical properties for diagram and icon types are `x_size` and `y_size`, which specify the horizontal and vertical dimensions

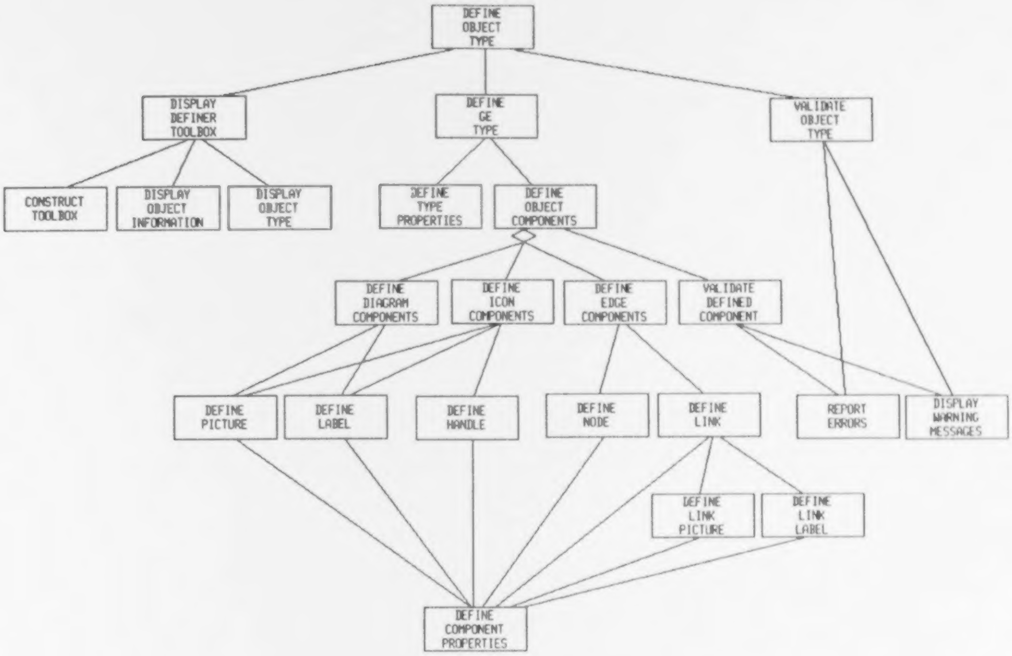

DEFINE OBJECT TYPE
DISPLAY DEFINER TOOLBOX
DEFINE GE TYPE
VALIDATE OBJECT TYPE
CONSTRUCT TOOLBOX
DISPLAY OBJECT INFORMATION
DISPLAY OBJECT TYPE
DEFINE TYPE PROPERTIES
DEFINE OBJECT COMPONENTS
DEFINE DIAGRAM COMPONENTS
DEFINE ICON COMPONENTS
DEFINE EDGE COMPONENTS
VALIDATE DEFINED COMPONENT
DEFINE PICTURE
DEFINE LABEL
DEFINE HANDLE
DEFINE NODE
DEFINE LINK
REPORT ERRORS
DISPLAY WARNING MESSAGES
DEFINE LINK PICTURE
DEFINE LINK LABEL
DEFINE COMPONENT PROPERTIES

Figure 16: Structure chart for "Define Object Type" process

of the object type. The graphical properties are usually defined using some graphic techniques such as *rubber-banding* and *drag-and-drop*. For instance, the size properties mentioned above are defined by dragging a "rubber-band" box on the screen to a desired size. However, there are certain types of properties that are not graphically definable. In those cases, GE Definer provides the user with other input methods involving use of selection dialogs, text-input boxes, etc.

- *Components*

  A set of graphical components is normally associated with each GE type. The types of components allowed depends on the object type being defined — a diagram type definer defines picture and label components; an icon type definer defines these two components plus handles; an edge type definer defines node and link components, and the link definer defines pictures and labels. Each of these component definers also define the component's own set of properties. When a component is defined, the object type definer also invokes a validation procedure to detect any errors and/or any missing parts. If problems are detected, error and/or warning messages are generated and displayed in a pop-up dialog window. The user has the choice of returning to that component definer and fixing the problems or ignoring the warnings.

Finally, when the user completes the definition of the object type with all of its components, GE Definer validates it, and in case of problems, warning messages are displayed.

The specifications of the current object type can be saved in two ways as depicted in Figure 13. The first approach allows the user to save the specifications in an Object File. This approach is appropriate for either a *partially* completed object type which requires further definition, or for a completed object type that is designed as a *library* object type for future reuse. In either case, the object type can be saved any time during the definition process.

The second approach allows the user to store the current GE type by appending it to the current environment. This process, however, requires that the object type itself is *completed* and *validated*, and does not produce any conflicts with the current environment. Once the object type is added to the current environment, the user can save the environment in order to store the definitions of that object type. The process that saves an environment involves generation of GE definitions. The GE definitions, which are described in the EDL/GE language, are generated by the GE Generator module. The module traverses the data structure of the current environment object, generates the GE definitions based on the stored specifications, and saves them into a specified Environment File.

In order to demonstrate how an actual graphical method modeling task is performed using GE Definer, a system "walk-through" of the tool is presented in Appendix C. The "walk-through" uses a real scenario of defining the "process" icon type of the Data Flow Diagraming method [MM85, Fin94a] to illustrate various features and functionality of GE Definer. Several screen dumps of the tool captured during the definition process are also presented to give the reader a thorough understanding of how GE Definer looks and works.

5.2 Conclusions

The present prototype version of GE Definer successfully accomplishes all the main objectives and design requirements described in Chapter 4. In short, the tool provides the method definer with an *intuitive, graphical* interface for defining *interactively* various graphical object types used in an SDM. Based on our past experience in modeling modern SDMs, such tool support greatly eases the method modeling task in Metaview. Complex graphical notations are typically used to represent model concepts and these graphical representations are easy to define *graphically* in the GE Definer.

In the following section, we discuss some important observations gained from the prototype version of GE Definer. These observations include the problems we have encountered during the implementation of GE Definer as well as an assessment of the usability and functionality of the prototype. Section 5.2.2 summarizes the contributions and the limitations of the current prototype in regard to graphical method modeling in Metaview. The section also suggests some future enhancements to GE Definer and proposes two research directions for improving tool support in graphical modeling.

5.2.1 Observations

During the implementation of GE Definer, we have encountered certain problems. We discuss three of them in this section because they are the important and more interesting ones. Two of these problems have already been solved in the current prototype version, but the other one requires further investigation. In the second half of this section, we discuss the results of GE Definer with respect to its usability and functionality. Several problems of the current prototype are also identified and described.

Problems Encountered

- *No unique identifier for each graphical element*

  Since every GE object type may have an arbitrary number of components of a certain type (such as pictures and labels), GE Definer must allow the user to specify which particular one of these components s/he wants to modified. Unfortunately, not every component type has a *unique* key to identify different component instances. Label is an example of such component types. Although a label component of a GE object type may be identified by the name of the attribute that it represents, this is not always a unique identifier because the same attribute can be represented by more than one label within a graphical object type.

In order to solve the problem, we added a browsing mode to the component "definers". With the browsing feature, the "definer" does not require the user to specify exactly which component object s/he wants to modify. Instead, the toolbox displays the definition of each existing component object one by one. When the user sees her/his desired one, s/he clicks on a button to select that object. Therefore, even though there is no way to identify uniquely a component object, GE Definer allows the user to browse through all the existing objects and select a desired one.

- *Difficulty in reusing code of other Metaview tools*

  The implementation of GE Definer uses some common modules, such as the EDL/GE Parser and the Symbol Table, and also reuses code of some existing Metaview tools. However, we found that these modules and code have quite different coding and documentation styles. As a result, we needed to spend more time and effort to understand these different pieces of code before we could successfully reuse them.

  In order to avoid such problem occur again in future tools' implementation, we designed some simple coding and documentation standards and applied them to GE Definer and other Metaview tools that were being developed (e.g., the Symbol Table). These standards will be constantly enhanced and extended to facilitate extensive code reuse for future Metaview tools.

- *Problem of building the GUI code in GE Definer*

  The GUI of GE Definer has certain interface components that are used frequently in the tool and are very similar in nature. For example, the toolboxes of various GE types' "definers" are very similar in their structures and functionality, and they are implemented using the same types of GUI components.

However, during the implementation of the prototype, we found that our original design of the GUI did not allow effective reuse of the defined interface components. In the prototype the GUI of GE Definer was built using traditional functional decomposition approach. If the various types of GUI components were implemented as object classes using object-oriented techniques, it appears that customization and reuse of these GUI objects would be much easier. Hence we expect that one of the future enhancements for GE Definer is to re-build its GUI using an object-oriented approach.

Usability and Functionality of the Prototype

Based on our observations on the usability and functionality of GE Definer, the prototype tool successfully provides an intuitive way for defining a method's graphical representations. The major benefit from GE Definer is its real-time, visual feedback. This feature makes the tool easy to use and thus reduces significantly the time and effort spent by the method definer in the modeling process. Its use of interactive dialogs and real-time validations also minimize errors made by the user.

On the contrary, we also identified several problems of the current prototype version. These observed problems are summarized as follows:

- *No definer for picture types*

  Presently, GE Definer does not have a facility for defining picture patterns. A tool for this purpose, called OGIPS, was already developed for the Metaview system, and we do not want our thesis research to overlap significantly with what had been done before. However, since the output of OGIPS is no longer usable by the current Metaview system (as described in Section 4.2.1), the picture types of a method must be hand-coded in GE language before GE Definer can use them to define the graphical representations of that method. Therefore there is still a need for extending GE Definer to include the capability of defining picture patterns.

- *On-line help facility not available yet*

In the current prototype version, the on-line help facility is not yet fully implemented. It is not a problem at this stage because the present users of GE Definer are mainly Metaview experts who have extensive knowledge on graphical method modeling using the system. However, in future, GE Definer will be used by other novice users, such as students and possibly industrial users, and at that time, the on-line help facility will be extremely important.

- *No use of colors*

  The current version of GE Definer does not support colors. This limitation makes the display of the graphical object type a little confusing because all the elements of the object type are displayed in black and white. It would be easier for the user to distinguish different graphical components if they were displayed in different colors.

- *Difficulty in fine-tuning some graphical properties*

  Although current prototype of GE Definer eases the definition of most graphical elements, it does not provide adequate support for defining some graphical properties in high precision. For example, GE Definer lets the user define the size of a label component by dragging a rubber-band box on the screen. This method sounds intuitive and efficient, but it does not easily produce an exact result meeting specific dimensions requirements. In other words, GE Definer lacks the support for "fine-tuning" some graphical properties.

- *Lack of support for defining symbolic constants and expressions*

  The graphical properties' values of an object type can depend on the values of other graphical types'. For example, it is common to define the location of an object type relative to the location of another. Thus, when the graphical definitions are hand-coded in the GE language, the method definer always defines a set of symbolic constants to hold the common values and uses them in the expressions to specify the values of certain properties. This approach is desirable because it makes the definitions easier to modify and more readable. Although GE Definer is capable of loading and using the pre-defined symbolic constants, the current prototype version does not support creation of new constants.

5.2.2 Summary of Results and Recommendations

In this section, we summarize the contributions and limitations of the current prototype of GE Definer with respect to graphical method modeling in Metaview. Recommendations are proposed to solve some of the existing problems and limitations.

Contributions

The present version of GE Definer has two major contributions to the modeling of graphical representations of an SDM in Metaview:

- *A graphical tool support for method modeling*

  GE Definer supports a graphical approach to define the graphical elements of a method. The success of GE Definer confirms that the graphical representations of a method is easy to define graphically, and much easier than previously tried textual approaches. This graphical tool can reduce the tedious method modeling task in Metaview and assist the method definer to produce a more reliable, error-free method model in less time and effort.

- *Generating graphical definitions in reusable form*

  GE Definer generates definitions that can be stored and reused by other tools of Metaview. Furthermore, these definitions are presented in a human readable format so that they can be reviewed by the method definer and other interested parties.

Limitations

Although GE Definer successfully provides automated and graphical support for defining *almost* all the graphical aspects of a method, its present version has limitations that make the tool *incomplete* in respect to the entire graphical modeling process required by Metaview. The two major limitations are:

- *Incomplete support for graphical method modeling*

  The current version of GE Definer, however, does not support definition of picture patterns nor graphical constraints.

- *Inadequate browsing facility for method definitions*

  Presently, GE Definer supports browsing on only the definitions and the "look" of the pre-defined GE types. It does not support other "views" of its defined graphical environment. Two examples of these possible *views* are the classes hierarchy of a specific graphical object type and the definitions of a graphical type in GE language.

To solve some of the existing problems and limitations, the following recommendations are proposed. These recommendations are divided into two groups — suggestions of enhancements and extensions to the current prototype and other proposed research directions.

Suggested enhancements and extensions to GE Definer

The enhancements and extensions that can make the current GE Definer a more powerful and user-friendly graphical modeling tool are:

- *Implement "Picture-Type Definer"*

  The definer of picture type should be implemented as a "definer" toolbox, similar to the ones existing in GE Definer, with buttons and controls for defining and editing the six types of geometrical primitives defined in the GE meta-model. The implementation of this definer may also reuse some of the graphic editing modules developed in OGIPS.

- *Re-build the GUI using Object-Oriented (O-O) Design*

- *Add on-line assistance facility*

  The current help facility should be extended so that it not only explains the features of the tools but also suggests modeling "tips" and/or "guidelines" to the user. Such assistance facility can guide the inexperienced user through the modeling process in a "step-by-step" approach. This feature is very much like the "wizard" program available in many commercial software systems.

- *Use color codings*
- *Add "fine-tuning" controls*

  There are at least two approaches to support fine-tuning in high precision. The first approach is to use a text input dialog to allow direct data input from the user. This approach is straight-forward and supports any level of precisions, but it may not be convenient to the user because it requires the use of keyboard. The second approach is to provide a "value adjuster" so that the user can click on the adjuster to increment or decrement a property's value in a specific interval.
- *Support the creation of symbolic constants*

  Symbolic constants are useful not only for defining the property values of a set of GE types, but also for generating more readable and concise GE definitions.

Research Directions

To provide a better and more complete tool support for graphical method modeling, there must be other tools to complement GE Definer. As described in Section 5.2.2, facilities are needed for browsing the definitions of a method environment in various *views* and for defining the graphical constraints of a method. The proposals of these two tools are important yet challenging in our Metaview system and require more investigations.

In the rest of this chapter, investigations of these two proposals are described. A proposed tool called "*Object Browser*" was designed for providing the environment browsing facility to the user. This tool is expected to complement GE Definer to support the graphical modeling task in Metaview. Implementation feasibility of a graphical tool for defining graphical constraints was also investigated. Some limitations of this approach are discussed at the end of this chapter.

5.3 Object Browser — The Design Proposal

During our research on graphical method modeling in Metaview, we identified the need for not only a graphical modeling tool but also an interactive method environment browser. Such a browser is expected to be capable of displaying the definitions of any modeled method in various types of "*views*" — each representing a particular aspect about a method environment. For example, one view may be a list of all conceptual and graphical object types defined in an environment, and another may be the class hierarchy of a specific icon type. This proposed graphical browser is very useful for method modeling in our Metaview system, and it can certainly complement our graphical modeling tool, GE Definer.

In Section 5.3.1, we discuss the motivation of the Object Browser proposal and the expected contributions of the browser to method modeling in Metaview. Section 5.3.2 and Section 5.3.3 summarizes the non-functional and functional requirements of Object Browser. Finally, Section 5.3.4 concludes the proposal of Object Browser and summarizes our contributions. In addition, the current state of the prototype version is described, and a recommendation on its future implementation is provided.

5.3.1 Motivation

The *Object Browser* is a Metaview tool that displays various kinds of information about the object types defined in an environment. In current version of Metaview, there are no such tools available, and the only way that a method definer can examine the defined object types is to read the textual EDL/GE definitions. This approach has drawbacks because the reader must have knowledge about EDL/GE language in order to understand the environment's definitions. In addition, some aspects of an object type are not easily recognized by simply looking at the textual definitions — for example, its graphical representation, its participation in a relationship type with other object types, and its role in the types hierarchy.

Hence we need an interactive, and preferably graphical, tool that can present the definitions of an object type (or of a set of object types) to the users. More importantly, the information of the object types can be presented in different views according to different users' demands. For example, two users can examine the same object type from two completely different views — one reviews the EDL/GE definitions of the object type, while the other looks at its graphical representation on the screen.

5.3.2 Non-Functional Requirements

There are several non-functional requirements that we need to consider when designing Object Browser:

- *User-friendliness*

  Since the target users of the tool are not necessarily experienced users of Metaview, Object Browser should be user-friendly in the sense that it makes good use of WIMPs and GUIs to ensure ease of user interactions. More importantly, the tool should display the information of the object types in various views in a neatly formatted and easily understandable manner. The tool should also provide the users with an extensive on-line help facility.

- *Compatibility*

  Because Object Browser is part of the Metaview system and is used normally with other tools, it should be as much compatible as possible with the current state of the system. The design of the tool should follow our system's presentation and implementation standards. For examples, Object Browser's user interface should offer the same "look and feel" as the other Metaview tools', and its implementation of the tool should use the same programming languages and coding standards. In addition, it should support the common data structures, repositories, and file formats used in the current Metaview system.

- *Interoperability*

  The Object Browser will often be used together with other Metaview tools, and therefore there is a requirement that these tools work cooperatively. This objective requires a well-designed interface within Object Browser so that it can communicate with the other tools efficiently and effectively.

- *Implementation Environment*

  Object Browser should be implemented using C++ and Motif toolkit because the tool uses the common data structure for environment representation (Symbol Table) which is implemented using C++ classes. Moreover, since all the current graphical tools available in Metaview use Motif to build their GUIs, the interface of the proposed browser should be implemented using the same toolkit in order to maintain a consistent "look and feel" with the other tools. In addition, the implementation of the GUI could also reuse some modules of the GE Definer. The tool runs under UNIX and X Window System on Sun workstations, which are the operation platforms for the current Metaview system.

5.3.3 Functional Requirements

In general, for the Object Browser to be a useful tool in Metaview, it must present different views of an environment according to the user's choice. The following functional requirements are necessary to achieve this general objective.

Input/Output

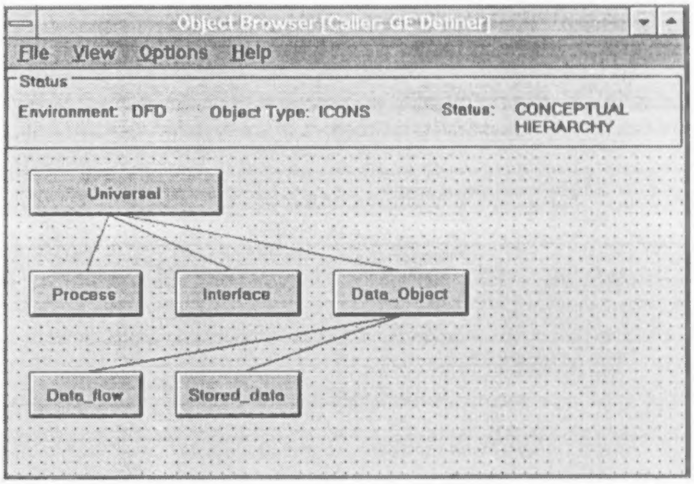
Object Browser must display the definitions of an object type, a set of object types, or all object types within an environment. To create a browser display, it has to retrieve the environment definitions from some sources. The two possible sources are the Symbol Table data structure and the EDL/GE definitions. In the former case, Object Browser can obtain the definitions of an environment or of a set of object types from another tool using the common data structure. For example, the method definer who is modeling a method using GE Definer may want to invoke Object Browser to review the classes hierarchy of the defined graphical object types. In this situation, Object Browser can request a copy of the data structure generated by GE Definer and display the specific type hierarchy on the screen. This approach is often used for the *incomplete* definitions of an environment or object type.

The second source, EDL/GE definitions, can be parsed by the EDL/GE Parser which is invoked by Object Browser. The environment definitions are stored in the Metaview Symbol Table data structure used by the browser so that it can retrieve specific information upon user's request. This approach is typically used for *completely* defined environments.

Object Browser can generate output that can be displayed or sent to other tools. These two output methods will be discussed at the end of this section.

User Interface

The user interface of Object Browser should look very similar to that of GE Definer. The GUIs of both tools are organized in three major parts as described below. Figure 17 shows the preliminary design of the GUI, prototyped using Visual BASIC.



Object Browser [Caller: GE Definer]

Status

Environment: DFD

Object Type: ICONS

Status: CONCEPTUAL HIERARCHY

Figure 17: The prototype GUI layout of Object Browser

- Menu Bar — It provides menus of commands to the user. The `FILE` menu consists of commands dealing with general input and output. The `VIEW` menu contains options to change the views of the information to be displayed. The `OPTIONS` menu contains other commands to configure the tool. Finally, the `HELP` menu consists of different options to activate an on-line help dialog.

- Status Panel — It displays the useful information for the current view. For examples, it shows the current environment's title, the kind of the current object type, the object type's name, and the selected view type.

- Display Area — This is the area where the information is displayed. In most cases, the display is graphical: for example, the graphical representation of an object type, the tree diagram of an object types' hierarchy, etc. Sometimes, however, the display can be textual, for example a table showing the names of all object types defined in the current environment.

Object Browser also has a title bar which displays not only its title but also the name of the tool that invokes the browser. This information reminds the user in which tool s/he is currently working.

Finally, Object Browser can also pop up dialog windows to show other, supplementary information. For example, when the user clicks on the object type displayed on the Main Display Area, a pop-up dialog presents its EDL/GE definition. Such pop-up dialogs allow the user to examine different types of information (*views*) of the same object simultaneously.

Information Presentations

Object Browser presents the information of the environment definition in various views. Some of these views express the relationships among a set of object types within an environment, while others describe the detailed definition of an object type or a component of that object type.

These views are represented graphically if possible. Some of the supported views are listed as follows:

- Lists of all conceptual (EARA) object types and/or graphical (GE) object types defined in an environment;

- Classification hierarchy of a specific object type;

- Aggregation hierarchy;

- Conceptual definition of an object type in table form (e.g., its type name, supertype name, attributes, etc.);
- Graphical representation and properties of a GE type (e.g., display of its picture pattern, locations of labels, etc.);
- EDL/GE Definitions of an environment or of an individual object type;

Object Browser should be able to display multiple views simultaneously.

Communications

One of the most important features of Object Browser is its communication capability with other tools in Metaview. This capability is useful for invoking Object Browser by another tool, for example GE Definer, and for transferring data back to the "calling" tool. For example, when a user opens a graphical object type in GE Definer, s/he can invoke Object Browser to show the object type's conceptual definition providing it exists. Another example occurs when the user needs to create a new definition (e.g., specify a type name) in GE Definer, s/he can call Object Browser and select a type name from it, and the name will be returned to GE Definer. In summary, a transparent communication capability is essential for the Object Browser to provide effective support for many of the other Metaview tools.

There are at least two approaches to implementing the communication mechanism. If Object Browser is implemented as an independent program, the communications between the browser and other tools can be handled using UNIX's sockets and command-line arguments [Hor86]. The sockets are interprocess communication channels that can be set up by the calling tool and Object Browser itself to transfer data in both directions. The command-line arguments can also be used to specify inputs, such as the filename of an environment file and the desired view type, to Object Browser. This input method is particularly useful when Object Browser is executed independently by the user.

The second approach is to share a common environment data structure (Symbol Table) between Object Browser and the other tool. Although this approach is a much simpler method, it can be used only if Object Browser is implemented as a library module and included in other tool's source code.

5.3.4 Conclusions

In summary, Object Browser is expected to offer significant contributions in two aspects of method modeling in Metaview:

1. It presents, graphically and interactively, object definition information for defined environments in different views. The various views of environment information helps the method definer to examine and verify different parts of the method specifications.
2. It provides interactive interface support when using other method modeling tools in Metaview. In other words, the browser complement other tools, for example GE Definer, to make the method modeling task more efficient and effective.

A prototype version of Object Browser is being implemented. However, only the main interface part of the tool has been built so far.

At the present stage, Object Browser is implemented as a common module so that other tools, for example GE Definer, can include it as part of their programs. This approach certainly simplifies the communication mechanism between the tools. Moreover, since the tools share the same data structure, the information retrieved by both Object Browser and the calling tool is guaranteed to be consistent and up-to-date. However, we recommend that further investigation should be done on the feasibility of building Object Browser as a standalone tool. One of the advantages of this approach is that the user will be able to execute Object Browser by itself. Another advantage is that the design of the tool will be more independent and flexible so that it can be extended or modified in future.

5.4 Investigation on a Graphical Constraint Definer

When we first proposed the idea of the graphical interface for modeling the graphical object types of a method, we also considered a similar graphical tool for defining the method's graphical constraints. There are basically two reasons that motivate an investigation of this approach to graphical constraint definition:

- The graphical representations and the graphical constraints of an SDM are the two aspects that form the *complete* graphical definitions of the method. These two aspects are closely related in the sense that the graphical representations model the notations used by a method and the graphical constraints define how these notations are used and presented. Thus it is desirable to provide the method definer with similar tool support to define these two aspects.
- Since both graphical representations and graphical constraints deal with graphic knowledge, our initial assumption was that both types of information could be defined more efficiently and effectively using a graphical tool.

After a more detailed investigation, we realized that most graphical constraints of the modern SDMs are too complex to be defined *clearly* and *precisely* using a graphical tool. Furthermore, even some very simple constraints — for example, that control the relative placement of a graphical type within a diagram — still cannot be unambiguously defined by a graphical tool because of its difficulty for associating precise semantics to the constraints. In general, a graphical tool cannot provide the *conciseness* and *precision* that a logical formula does, when defining a graphical constraint.

In this section, we discuss the potential limitations for defining graphical constraints *graphically* and illustrate our arguments using some examples based on the Data Flow Digraming method. The conclusions of our investigation and suggestions for two future research directions are then presented.

5.4.1 Limitations for Defining Graphical Constraints Graphically

The graphical constraints define precisely the rules that a specific method has to follow when presenting their model concepts. There are basically two major characteristics of these constraints. First, the graphical constraints of an SDM are extremely diverse in complexity and scope. Second, each of these constraints must have a precise and unambiguous semantics. In the rest of this section, we discuss our investigation of these characteristics and analyze the potential limitations of a graphical tool in this regard.

Diversity

The graphical constraints used vary greatly from method to method; even the constraints used in the same method may be very different in complexity and scope among themselves. For example, a representational constraint may be as simple as a mathematical formula that determines the placement of an icon within a diagram, but may also be as complex as a logical rule that defines a set of cases (conditions) in which an icon can be used. Furthermore, the scope of the constraints may also vary greatly ranging from an individual graphical type to all object types within an environment.

Unfortunately, there are no accepted models or rules for defining all the different types of representational constraints of a diagraming method. Some rules that sound common and reasonable in one method may not make sense in another. Consider examples of two *structured methods*: Structure Chart and Data Flow Diagraming (DFD): The Structure Chart method controls the layout of a diagram such that the icons representing sub-modules have to be positioned *below* the icons representing modules in a upper level [MM85]. The DFD method does not impose such explicit rules on its diagram layout[4].

One of the most important characteristics of a successful constraint defining interface is its *flexibility*. The interface has to be flexible enough to support a wide variety of rules including user-defined ones. The new version of ECL (Environment Constraint Language) in Metaview, also called ECL III [Fin94c], is a flexible language to describe the graphical constraints. The language has a rich set of language constructs for building logical rules and predicates; for examples, it supports *universal* and *existential quantifiers* [GN88] to deal with different scopes of constraints. It also provides arithmetical functions to handle rules that involve mathematical formulae and calculations.

---

[4] Although rules can be added to ensure a better readability of a DFD diagram, for example moving all the *terminators* towards the boundaries of the diagram, they are not part of the method.

It is difficult for a graphical tool to offer such a level of flexibility. It is not easy to extend a graphical tool dynamically to support a set of user-defined rules, because such extensions require new support of not only the *concepts* of the rules but also the *graphical techniques* to define those rules. Therefore, the tool must support a large class of graphical constraints to assist the method definer in creating her/his constraint rules that are appropriate for a given method. This approach leads to the efficiency problem — there is no efficient way yet to classify and support all the possible graphical constraints by a graphical tool because there are so many various types of constraints. Hence, it is impossible for a tool to capture all the possible semantics of the graphical constraints and allow the user to assign semantics to her/his defined constraint without ambiguity. In the following subsection, the problem of ambiguous semantics is discussed in greater detail.

Precise Semantics

The graphical constraints must be defined precisely and unambiguously, because they define the manner in which the model concepts are represented diagrammatically. In general, the representational constraints can be classified into two categories:

- rules that specify the relationships between graphical object types within a diagram;
- rules that define the "look" of a set of graphical types in the diagram.

In the former case, they are not intuitively definable by a graphical tool because they deal with the concepts *behind* the method's representations. An example[5] constraint of this category for the DFD method is that each *data element* may be used only once in an edge (*data flow*). The definition of this constraint in ECL III is shown as follows:

---

[5] This example and the one discussed below are cited from [Fin94a].

```
CONSTRAINT  no_multiple_use_of_data (GRAPHICAL diag : any_level)
ALL flow0 FROM {flow @ diag}
    ALL flow1 FROM {flow @ diag : *->data == flow0->data}
        SATISFY flow1 == flow0
OTHERWISE
    message {"Warning..."};
```

Although reader, who is not familiar with the ECL language, may not be able to understand the defined rule at first glance, this constraint definition is certainly a concise and unambiguous way to specify the above-mentioned example. On the contrary, rules of this type are not effectively nor intuitively definable using a graphical interface.

The second category of graphical constraints, those responsible for defining the presentation of the graphical objects in a diagram, ensure that the resulting model representations look "as expected". An inherent limitation of a graphical tool is its inability to define, or present, semantics in the form of logical relationship (i.e., characteristic functions). That is, when a graphical tool displays graphical symbols on the computer screen, the tool does not give the user a clear meaning on how the presentations of these objects relate and affect one another.

A further example of a graphical constraint in the DFD method is that the icon of the *data element* passed by a *data flow* must be located "near" the corresponding edge, where the meaning of the adjective "near" represents a reasonable distance between the icon and the edge. This constraint is defined in ECL III as follows:

```
CONSTRAINT  data_close_to_the_edge (GRAPHICAL diag : any_level)
ALL f FROM {flow @ diag}
WHERE d = f->data,
      xpos = _origin(d) + _width(d)/2,
      ypos = _top(d) + _height(d)/2
SATISFY _edge_distance(f, xpos, ypos) <= 50
OTHERWISE
      message ("Warning...");
```

This constraint definition specifies that the *shortest* distance (returned by the function `_edge_distance`) between the center of the icon representing the data element and its corresponding edge has to be less than or equal to 50 pixels (this number is arbitrarily chosen); otherwise a warning message is printed.

To define this rule graphically, the user would likely select the two involved graphical object types and arrange them on the screen in attempt to define such a placement constraint between them. However, the problem is that it is difficult to define graphically the precise semantics for a general constraint such as "nearness" in this example — the tool may interpret this constraint in various ways. Some alternative interpretations are: *the "data" icon must be away from the "flow" edge* no less than *50 pixels? The icon must be above/below the edge? The icon must be on the left/right side of the edge? The icon must always maintain the same distances from the two ends of the edge?*

To provide an ability to define precisely general graphical constraints in a graphical manner, the tool must provide an extensive set of logical predicates such that the user could precisely define what the various notions of "nearness" mean. Because graphical constraints of modern SDMs are extremely diverse, so far there does not appear to be an efficient and effective way to classify and support them in a graphical tool.

5.4.2 Conclusions

In summary, our preliminary investigation of the feasibility of a graphical tool for defining graphical constraints indicates that it is difficult for such a tool to define concisely and precisely the graphical constraints. The major problem is the limitation of a graphical tool in supporting the two characteristics of the graphical constraints of modern SDMs: *diversity* and *precise semantics*. A graphical tool can provide a concise way to present and define the *appearance* of graphical symbols but not the *constraints* among them. We believe that a description language supporting logic formulae and predicates, for example ECL III, can do a better job than a graphical tool because the language produces effectively more concise and accurate constraint definitions. More examples of definition of graphical constraints in ECL III can be founded in [Fin94a, Zhu94].

On the other hand, we agree that novice users may find it difficult to define or review the graphical constraint definitions using ECL. It is because the grammar of the language is quite complicated and requires the new user to spend some time to be familiar with it. Since our fundamental goal of this thesis research is to propose interactive, and preferably graphical, tool support for the method modeling process in Metaview, we here suggest two possible future research directions:

1. *Interactive Editor for ECL*

   Even though a graphical tool may not help effectively the method definer to define graphical constraints, an interactive editor for ECL should be helpful in reducing significantly the time and effort spent by the user, especially novice user who is not familiar with the language.

   Our preliminary idea is that the editor has a GUI that provides interactive dialogs through which the user can enter logic formulae that define the graphical constraints. These dialogs should take care of the grammar of ECL language and require the user to input only the predicates and the formulae. Furthermore, the editor should also provide on-line help facility to guide the user to complete the definition of the constraints. We recommend that such ECL editor should be designed to work cooperatively with GE Definer in order to provide a *complete* tool support for graphical method modeling in Metaview.

2. *Further Investigation on Graphical Constraint Definer*

   Although our investigation realized the limitations of a graphical tool for defining graphical constraints, further investigation on this topic is still challenging and promising. As reported in the previous section, the major problem in the implementation of such a constraint definer is that there is a huge variety of graphical constraints used in modern SDMs. We do not as yet have any efficient and effective model or classification scheme to support these different kinds of constraints. Thus, more research is recommended in this area, and hopefully some day, a graphical tool can be built which can define any kind of representational constraint in a concise and precise manner.

Chapter 6

Conclusions and Future Research

Method modeling is a common and important feature of many CASE shells, because it captures the necessary knowledge about a software development method in order to generate a *customized* CASE environment. Software development method modeling is a challenging task, especially when today's methods adopt complex models and represent their modeling concepts using a large variety of graphical notations and techniques. A deficiency in method modeling in most meta-CASE or CASE shell systems is the lack of an efficient and effective approach for defining a method's graphical representations. This thesis research focuses on addressing this deficiency.

The objectives of this research were to:

- design a textual description language for defining the graphical representations of a method;
- investigate a graphical approach to graphical method modeling;
- propose and prototype a graphical tool to support the graphical method modeling task;
- identify the requirements and limitations of the graphical modeling tool and recommend directions of future research in this area.

Section 6.1 summarizes the results of this research and discusses its major contributions. Section 6.2 suggests future research directions.

6.1 Conclusions

The results of this research are concluded in the following two sections which summarize the major contributions and discuss the limitations of the prototype graphical modeling tool.

6.1.1 Achievements and Contributions

Our research successfully fulfilled the thesis objectives with the following achievements:

- *Designed GE language*

  To provide basic support for the definition of a graphical method model, a textual description language, called GE language, was designed. The language is used in Metaview as a *medium* to store and transfer the graphical specifications of a method between tools and to publish the definitions for verification and review purposes.

- *Investigated Graphical Approach to Graphical Method Definition*

  To ease the tedious graphical method modeling task, a graphical approach to graphical method modeling was proposed. Investigation of the approach identified various requirements and problems that were used as a basis for the design of a graphical tool.

- *Prototyped GE Definer*

  A prototype of the graphical modeling tool, GE Definer, was constructed. The tool provides intuitive and effective support for defining the graphical elements used in a method. More importantly, the results of the prototype activity identified a number of benefits and limitations of the current approach. Based on this analysis several avenues for future research are identified.

The achievements of this thesis research resulted in a number of significant contributions with respect to both the study of the graphical method modeling problem and our Metaview meta-CASE system. Our investigation of the graphical approach to the graphical modeling problem identified several requirements and benefits of the interactive, graphical modeling tool, including the need for user-friendly interfaces, visual feedback, real-time interactions, and information browsing. Initial observations of our work indicate that this graphical tool approach can reduce significantly the time and effort spent, and also minimize errors made by the method definer in the method modeling task.

The GE language and the GE Definer implemented in this research greatly extends the method modeling capability of Metaview. Nevertheless, their current capabilities are limited and not without problems and, as a result, a number of future research directions for Metaview are suggested. These suggestions will be discussed at the end of this chapter.

6.1.2 Limitations

Although our research objectives are achieved, the current prototype has limitations that should be investigated further. The two major limitations are summarized as follows:

- *Incomplete Graphical Modeling Support*

  The present version of GE Definer provides intuitive support for defining various graphical types of a method environment; however, it is still not a *complete* tool for graphical method modeling in Metaview. In particular, it lacks support of the definition of picture patterns and graphical constraints. The prototype version also has minor problems as identified in 5.2.1 that require further enhancements and extensions.

- *Inadequate Browsing Facility*

  GE Definer supports browsing on only the definitions and the "look" of the defined graphical types. Such a browsing capability is too limited because it does not provide the method definer with multiple *views* of the method environment definitions. An ideal browsing facility should allow the user to review the conceptual definitions of object types during the graphical modeling process.

6.2 Suggestions of Future Research

Several promising and challenging areas of future research and development related to CASE environment definitions in Metaview include:

- *Enhancements to GE Definer*

  As mentioned in Section 6.1.2, the present prototype version of GE Definer requires enhancements and extensions to provide mechanism for defining picture patterns and other features, such as on-line assistance and color codings.

- *Implementation of Object Browser*

  More work is required to complete the existing design and prototype of the Object Browser. In the future, further effort should be spent to extend the tool's design. One potential extension, as described in Section 5.3.4, is to enhance Object Browser's GUI and communication facility so that it can operate as a stand-alone tool.

- *Tool Support for Defining Graphical Constraints*

  Our investigation, as discussed in Section 5.4, identified the potential problems of using a graphical approach to the definition of graphical constraints. Nevertheless, additional tool support for defining graphical constraints is desired for Metaview users because the constraint definition language, ECL III, albeit powerful, is not easy to learn and use. Therefore, we suggest that an interactive editor of ECL language be implemented as an interim solution to the problem. In long term, further research on the graphical tool support should be completed.

- *Conditional Representations*

  In some diagraming methods, such as [Boo91], the representations of the object types may change according to their property values. For example, the picture patterns used by an object type may vary slightly to reflect value changes of its properties. To support such conditional representations for the graphical object types, *conditional properties* must be supported by the underlying meta-model. In fact, this feature has already been proposed in Metaview as one of the future extensions to the GE metamodel.

  If this feature is to be supported, however, our graphical modeling tool must be extended accordingly. For example, to allow input of conditional properties for the GE types, GE Definer should provide an interactive dialog to capture the "conditional formulae". These formulae may be as simple as "*IF-THEN-ELSE*" conditions but may also be as complex as compound logical formulae. Thus, supporting these conditional formulae requires further investigations and extensions to GE Definer.

- *Supporting other kinds of representations*

  Although a majority of modern software development methods uses graphs and diagrams to represent the methods' model concepts, there are still many commonly used methods that adopt other kinds of representation techniques, such as tables[1], matrices[2], charts[3], and more recently, multimedia and hypertext [LKK$^+$94]. Thus, in order to support a larger variety of graphical methods, it is important to extend Metaview's graphical modeling capability. Future research should be done on extensions to not only the GE meta-model but also the graphical tool support.

- *A Graphical Interface for Conceptual Modeling in Metaview*

  Due to our success in using a graphical approach to the definition of graphical representations, we are encouraged to investigate the feasibility of applying this approach to conceptual modeling. Some existing CASE shells, such as MetaEdit [LKK$^+$94], have graphical interfaces to define the model concepts of a method. The MetaEdit's approach [LST$^+$91, Ros95] represents its meta-model's conceptual types in various graphical symbols and uses a graphical interface to manipulate these symbols to build *graphically* the model of a method.

  Because Metaview's conceptual meta-model, EARA, is an extension of the Entity-Relationship (ER) data-model [Che76], it is feasible that the method concepts modeled by EARA meta-model can be represented using some "ER-*like*" diagraming notations and techniques. To achieve this, further research is required to investigate how EARA's *aggregation* and *classification* features can be modeled graphically. In short, a graphical tool support for conceptual modeling is desirable in Metaview because it is consistent and more convenient to have similar tools support both conceptual and graphical method definitions.

---

[1] e.g., decision tables [MM85].

[2] e.g., Andersen Consulting's Method/1 [AAC87].

[3] e.g., Gantt (Timeline) Charts [Pre92].

Bibliography

[AAC87] Arthur Andersen Consulting. *Foundation-Method/1: Documentation*, 1987. Version 8.0, Chicago.

[BBD⁺89] P. Bergsten, J. Bubenko, R. Dahl, et al. RAMATIC — a CASE shell for implementation of specific CASE tools. Technical Report TEMPORA T6.1, Swedish Institute for Systems Development (SISU), Stockholm, 1989.

[BCM⁺94] A. W. Brown, D. J. Carney, E. J. Morris, et al. *Principles of CASE Tool Integration*. Oxford University Press, 1994.

[BEM92] A. W. Brown, A. N. Earl, and J.A. McDermid. *Software Engineering Environment — Automated Support for Software Engineering*, chapter 1–3 & 11–12. McGraw-Hill Book Company, 1992.

[BKM90] S. Burson, G. B. Kotik, and L. Z. Markosian. A program transformation approach to automating software re-engineering. In *Proc. of the 14th Annual International Computer Software and Applications Conference*, pages pp.314–322, October 1990.

[Boo91] G. Booch. *Object Oriented Design with Applications*. Benjamin/Cummings Publishing Co., Inc., Redwood City, CA, 1991.

[BW] S. Brinkkemper and G. M. Wijers. Customizable CASE-tools: A need for effective automated support. Software Engineering Research Centre, the Netherlands.

[BWS87] S. Beer, R. Welland, and I. Sommerville. Software design automation in an IPSE. In H. K. Nichols and D. Simpson, editors, *Proc. of First European Software Engineering Conference ESEC'87*, pages pp.89–97. Springer-Verlag, September 1987.

[Cha90] S.-K. Chang (ed). *Visual Languages and Visual Programming*. Plenum Press, New York, 1990.

[Che76] P. P. Chen. The entity-relationship model — toward a unified view of data. *ACM Transactions on Database Systems*, 1(1):pp.9–36, March 1976.

[Che83] P. P. Chen. ER — a historical perspective and future directions. In C. G. Davis, editor, *Entity Relationship Approach to Software Engineering*. North-Holland, Amsterdam, 1983.

[CN89] M. Chen and J. F. Nunamaker Jr. Metaplex: an intergrated environment for organization and information systems development. In *Proc. of Tenth International Conference on Information Systems*, pages pp.141–151, Boston, December 1989.

[Con94] S. Conger. *The New Software Engineering*. Wadsworth Publishing Co., Belmont, CA, 1994.

[CR88] E. J. Chikofsky and B. L. Rubinstein. CASE: reliability engineering for information systems. *IEEE Software*, pages pp.11–16, March 1988.

[CTI93] Cadre Technologies Inc. *Ensemble System Understanding User's Guide*, 1993. Release 5.0.

[DEF+87] S. A. Dart, R. J. Ellison, P. H. Feiler, et al. Software development environments. *Computer*, 20(11):pp.18–28, November 1987.

[DST89] J. M. DeDourek, P. G. Sorenson, and J.-P. Tremblay. Metasystems for information processing system specification environments. *INFOR*, 27(3):pp.311–337, August 1989.

[Fin93a] P. Findeisen. Environment definer guide. Department of Computing Science, University of Alberta, March 1993.

[Fin93b] P. Findeisen. Environment definition for database engine. Department of Computing Science, University of Alberta, May 1993.
URL: http://web.cs.ualberta.ca/~softeng/Metaview/doc/dbeng_reqs.ps.

[Fin93c] P. Findeisen. The graphical extension for the EARA model. Department of Computing Science, University of Alberta, May 1993.
URL: http://web.cs.ualberta.ca/~softeng/Metaview/doc/graph_ext.ps.

[Fin93d] P. Findeisen. MGED reference manual (motif version). Department of Computing Science, University of Alberta, May 1993. Release 2.0.
URL: http://web.cs.ualberta.ca/~softeng/Metaview/doc/mged.ps.

[Fin94a] P. Findeisen. A complete definition of data flow diagram environment for Metaview. Department of Computing Science, University of Alberta, July 1994.
URL: http://web.cs.ualberta.ca/~softeng/Metaview/doc/dfd_defn.ps.

[Fin94b] P. Findeisen. The EARA model for Metaview — a reference. Department of Computing Science, University of Alberta, June 1994.
URL: http://web.cs.ualberta.ca/~softeng/Metaview/doc/eara_model.ps.

[Fin94c] P. Findeisen. Environment constraint language for Metaview. Department of Computing Science, University of Alberta, May 1994. Draft Proposal.
URL: http://web.cs.ualberta.ca/~softeng/Metaview/doc/ecl_man.ps.

[Fin94d] P. Findeisen. The Metaview system. Department of Computing Science, University of Alberta, June 1994.
URL: http://web.cs.ualberta.ca/~softeng/Metaview/doc/system.ps.

[Fin94e] P. Findeisen. Project daemon reference manual. Department of Computing Science, University of Alberta, August 1994. Version 2.3.
URL: http://web.cs.ualberta.ca/~softeng/Metaview/doc/pd_man.ps.

[Fin94f] P. Findeisen. Re-designing ECL for Metaview. Department of Computing Science, University of Alberta, March 1994.

[Fin95] P. Findeisen. Environment representation for Metaview tools. Department of Computing Science, University of Alberta, August 1995. Draft v.1.3.

[FK92] R. G. Fichman and C. F. Kenerer. Object-oriented and conventional analysis and design methodologies: Comparison and critique. *Computer*, 25(10):pp.22–39, October 1992.

[FW90] R. T. Fleming and N. Wybolt. CASE tool frameworks. *UNIX Review*, 8(12):pp.24–32, 1990.

[Gad93] D. Gadwal. ECL user's manual. Department of Computational Science, University of Saskatchewan, November 1993.

[GFS$^+$94] D. Gadwal, P. Findeisen, P. G. Sorenson, et al. Generating customizable software specification environments using metaview. Technical Report TR 94-2, Department of Computational Science, University of Saskatchewan, 1994.
URL: http://web.cs.ualberta.ca/~softeng/Metaview/doc/wmrt_defn.ps.

[GH89] G. H. Galal and P. A. V. Hall. Computer-aided software engineering. *Computer-Aided Engineering Journal*, 6(4):pp.113–120, August 1989.

[GLM94] D. Gadwal, P. Lo, and B. Millar. EDL/GE user's manual. Department of Computing Science, University of Alberta and Department of Computational Science, University of Saskatchewan, June 1994.
URL: http://web.cs.ualberta.ca/~softeng/Metaview/doc/edl_man.ps.

[GM93] D. Gadwal and B. Millar. EDL system's manual. Department of Computational Science, University of Saskatchewan, January 1993.

[GN88] M. R. Genesereth and N. J. Nilsson. *Logical Foundations of Artificial Intelligence*. Morgan Kaufmann Publishers, Inc., 1988.

[HO93] M. Heym and H. Österle. Computer-aided methodology engineering. *Information and Software Technology*, 35(6/7):pp.345–354, 1993.

[Hor86] R. N. Horspool. *C Programming in the Berkeley UNIX Environment*. Prentice-Hall, 1986.

[ITCa] *Customizer Reference Guide*. Cambridge, USA, 1987 edition.

[ITCb] *Excelerator Reference Guide*. Cambridge, USA, 1987, 1990 edition.

[JJL+93] D. R. Jeffery, J. O. Jenkins, G. C. Low, et al. An empirical evaluation of the use of CASE tools. In *Proc. of Sixth International Workshop on Computer-Aided Software Engineering*, pages pp.81–86, 1993.

[KP91] G. E. Kaiser and D. E. Perry. Models of software development environments. *IEEE Transactions on Software Engineering*, 17(3):pp.283–295, March 1991.

[KS94] S. Kelly and K. Smolander. Evolution and issues in metaCASE. Department of Computer Science and Information Systems, University of Jyväskylä, Finland, 1994.

[LKK+94] K. Lyytinen, P. Kerola, J. Kaipala, et al. MetaPHOR: Metamodeling, principles, hypertext, objects and repositories. Technical Report TR-7, Department of Computer Science and Information Systems, University of Jyväskylä, Jyväskylä, Finland, December 1994.

[LMR+92] K. Lyytinen, P. Marttiin, M. Rossi, et al. Modeling requirements for future CASE: issues and implementation considerations. In *Proc. of The 13th International Conference on Information Systems*, Dallas, Texas, December 1992.

[LST+91] K. Lyytinen, K. Smolander, V.-P. Tahvanainen, et al. MetaEdit — a flexible graphical environment for methodology modelling. In *Advanced Information System Engineering: Proc. of Third International Conference of CAiSE'91*, Trondheim, Norway, May 1991. Springer-Verlag, Berlin.

[Mac91] C. MacKenzie. OGIPS user's manual. Department of Computational Science, University of Saskatchewan, July 1991.

[McA88] A. J. McAllister. *Modeling Concepts For Specification Environments*. PhD thesis, Department of Computational Science, University of Saskatchewan, Saskatoon, March 1988.

[McC89] C. McClure. *CASE is Software Automation*. Prentice Hall, Englewood Cliffs, 1989.

[MK88] H. A. Müller and K. Klashinsky. Rigi — a system for programming-in-the-large. In *Proc. of the 10th International Conference on Software Engineering (ICSE)*, pages pp.80–86. IEEE Comp. Soc. Press, April 1988.

[MM85] J. Martin and C. McClure. *Diagramming Techniques for Analysts and Programmers*. Prentice-Hall, Englewood Cliffs, New Jersey, U.S., 1985.

[MM87] D. A. Marca and C. L. McGowan. *SADT Structured Analysis and Design Technique*. McGraw-Hill, 1987.

[MRT$^+$93] P. Marttiin, M. Rossi, V.-P. Tahvanainen, et al. A comparative review of CASE shells — a preliminary framework and research outcomes. *Information and Management*, 25:pp.11–31, 1993.

[MTO$^+$92] H. A. Müller, S. R. Tilley, M. A. Orgun, et al. A reverse engineering environment based on spatial and visual software interconnection models. In *SIGSOFT'92: Proc. of the Fifth ACM SIGSOFT Symposium on Software Development Environments*, pages pp.88–98, December 1992. In ACM Software Engineering Notes.

[Orv92] W. J. Orvis. *Do It Yourself Visual BASIC*. Programming Series. Sams, Carmel, Ind., 1st edition, 1992.

[OSF92] *OSF/Motif Style Guide*. Cambridge, MA, USA, August 1992. Revision 1.2.

[Ped92] J. Peddie. *Graphical User Interfaces and Graphic Standards.* McGraw-Hill, Inc., 1992.

[Pre92] R. S. Pressman. *Software Engineering — A Practitioner's Approach.* McGraw-Hill, 3 edition, 1992.

[PVU77] J. Peeck, G. Van Dam, and R. W. Uhlenbeck. Incidental cues and picture/word differences in recall. *Perceptual and Motor Skills*, 45(3):pp.1211–1215, 1977.

[R+91] J. Rumbaugh et al. *Object-Oriented Modeling and Design.* Prentice Hall, 1991.

[RGS+92] M. Rossi, M. Gustafsson, K. Smolander, et al. Metamodelling editor as a front end tool for a CASE shell. In *Advanced Information System Engineering: Proc. of Fourth International Conference of CAiSE'92*, Berlin, May 1992. Springer-Verlag.

[Ros95] M. Rossi. The MetaEdit CAME environment. Department of Computing Science and Information Systems, University of Jyväskylä, 1995.

[SM91] S. Shlaer and S. J. Mellor. *Object Lifecycles — Modeling the world in states.* Youdon Press, 1991.

[STM88] P. G. Sorenson, J.-P. Tremblay, and A. J. McAllister. The Metaview system for many specification environments. *IEEE Software*, 30(3):pp.30–38, March 1988.

[Str87] B. Stroustrup. *The C++ Programming Language.* Addison-Wesley, 1987.

[tHNV+92] A.H.M. ter Hofstede, E. R. Nieuwland, T. F. Verhoef, et al. Integrated specification of method and graphic knowledge. Software Engineering Research Centre (SERC), the Netherlands, February 1992. SOCRATES project.

[tHVW91] A.H.M. ter Hofstede, T. F. Verhoef, and G. M. Wijers. Structuring modelling knowledge for CASE shells. In *Advanced Information System Engineer-*

ing: Proc. of Third International Conference of CAiSE'91, pages pp.502–524, Trondheim, Norway, May 1991. Springer-Verlag.

[Was90] A. I. Wasserman. Tool integration in software engineering environments. *Electro/90 Conference Record*, pages pp.419–425, 1990.

[Wel92] R. J. Welke. The CASE repository: More than another database application. In *Challenges and Strategies for Research in Systems Development*. Wiley, Chichester, UK, 1992.

[You89] E. Yourdon. *Modern Structured Analysis*. Prentice Hall, Englewood Cliffs, 1989.

[You90] D. A. Young. *The X Window System Programming and Applications with Xt (OSF/Motif Edition)*. Prentice Hall, 1990.

[Zar90] P. F. Zarrella. CASE tool integration and standardization. Technical Report CMU/SEI-90-TR-14, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, Pennsylvania, December 1990.

[Zhu94] Y. Zhuang. Object-oriented modeling in Metaview. Master's thesis, Department of Computing Science, University of Alberta, 1994.
URL: http://web.cs.ualberta.ca/~softeng/Thesis/yong.ps.

Appendix A

Summary of GE Language Syntax

The formal syntax of GE language is summarized in this appendix. The syntax is described in *augmented* BNF, which is briefly introduced in Section A.1. Section A.2 summarizes the symbols used in GE language.

A.1 Augmented BNF

Augmented BNF includes the following constructs (the first two are part of extended BNF and the third is a new construct):

$[z\ ]$ means zero or *one* instance of z is required;

$\{z\}$ means zero or *more* instances of z are required. This construct may be enhanced by the use of a subscript to specify a minimum value and/or a superscript to specify a maximum value. For example, $\{\ z\ \}_1^6$ means that there must be between one and six instances of z;

$\prec a \mid \ldots \mid n \succ$ is used to provide a simple and efficient means for specifying a set of *order-independent* options. Every item within the $\prec\succ$ still follows the extended BNF specification. A simple example is the statement $\prec A \mid B \mid C \succ$. It means that there is one instance of each A, B and C but the order in which they appear doesn't matter. Thus the valid instantiations of this statement are as follows:

A B C
B A C
C A B
A C B
B C A
C B A

Adding the extended BNF options of square brackets to form the statement $\prec [A] \mid [B] \mid [C] \succ$ increases the complexity of specifying the valid options as there may be zero or one of A, B and C. Thus now there are 16 valid instantiations. They are:

nil
A
A B
A C
A B C
A C B
B
B A
B C
B A C
B C A
C
C A
C B
C A B
C B A

A.2 Symbols Used in GE Language

A number of symbols are used in the GE language as either *operators* or *separators*. To avoid confusion, each of these symbols is assigned a unique name, and they are used in the formal specification of the language in the next section. Table A.3 summarizes these symbols and their associated names.

Name	*Symbol*
ASTERISK	*
BAR	|
COLON	:
COMMA	,
DD	..
DQUOTE	”
EQUAL	=
LB	[
LP	(
PERIOD	.
QUOTE	’
RB	]
RP	)
SEMI	;

Table A.3: Symbols used in GE language

A.3 Formal Syntax of GE Language

Graphical Properties

```
<properties_clause>   ::=  PROPERTIES LP [ <property> {COMMA <property>} ] RP
<property>            ::=  <property_name> EQUAL <property_value>
<property_name>       ::=  <identifier>
<property_value>      ::=  <identifier>
```

Picture Primitives

```
<point_primitive>     ::=  POINT  ≺      AT <coordinates>
                                     |   [<properties_clause>]
                                  ≻

<line_primitive>      ::=  LINE   ≺      FROM <coordinates>
                                     |   TO <coordinates>
                                     |   [<properties_clause>]
                                  ≻

<arc_primitive>       ::=  ARC    ≺      CENTER <coordinates>
                                     |   <radius_clause>
                                     |   START <angle>
                                     |   SPAN <angle>
                                     |   [<properties_clause>]
                                  ≻

<circle_primitive>    ::=  CIRCLE ≺      CENTER <coordinates>
                                     |   RADIUS <radius>
                                     |   [<properties_clause>]
                                  ≻

<box_primitive>       ::=  BOX    ≺      FROM <coordinates>
                                     |   TO <coordinates>
                                     |   [<radius_clause>]
                                     |   [<properties_clause>]
                                  ≻

<text_primitive>      ::=  TEXT   ≺      AT <coordinates>
                                     |   [<properties_clause>]
                                  ≻
```

```
<angle>              ::=  <integer between 0 and 360>
<radius_clause>      ::=  RADIUS <radius> | RADIUSES <radius> COMMA <radius>
<radius>             ::=  <integer>
<coordinates>        ::=  LP <integer> COMMA <integer> RP
<location>           ::=  <coordinates> | <identifier>
```

Picture Type

```
<picture_type_def>   ::=  PICTURE_TYPE <identifier> <picture_def_body>

<picture_def_body>   ::=  <picture_type_spec> <picture_primitives>

<picture_type_spec>  ::=  ≺        [GENERIC]
                            |      [IS_A <identifier>]
                          ≻

<picture_primitives> ::=  ≺        {<point_primitive>}
                            |      {<line_primitive>}
                            |      {<arc_primitive>}
                            |      {<circle_primitive>}
                            |      {<box_primitive>}
                            |      {<text_primitive>}
                          ≻
```

Picture Component

```
<pictures_clause>    ::=  PICTURES LP [ <picture_spec> {COMMA <picture_spec>} ] RP

<picture_spec>       ::=  <identifier>  ≺        AT <location>
                                          |      ROTATED <angle>
                                          |      [<properties_clause>]
                                        ≻
```

Label Component

```
<labels_clause>      ::=  LABELS LP [ <label_spec> {COMMA <label_spec>} ] RP

<label_spec>         ::=  <identifier>  ≺        AT <location>
                                          |      [<properties_clause>]
```

```
≻
```

Diagram Type

```
<diagram_type_def>     ::=  DIAGRAM_TYPE <identifier> <diagram_def_body> SEMI

<diagram_def_body>     ::=  <diagram_spec> <diagram_components>

<diagram_spec>         ::=  ≺       [GENERIC]
                                |   [IS_A <identifier>]
                            ≻

<diagram_components> ::=  ≺       [<properties_clause>]
                                |   [<pictures_clause>]
                                |   [<labels_clause>]
                            ≻
```

Handle Component

```
<handles_clause>       ::=  HANDLES LP [ <handle_spec> {COMMA <handle_spec>} ] RP

<handle_spec>          ::=  <relationship_name> PERIOD <role_name>
                                AT LP <range> {COMMA <range>} RP
                                [<properties_clause>]

<relationship_name>    ::=  ASTERISK | <identifier>
<role_name>            ::=  ASTERISK | <identifier>
<range>                ::=  LP <x_range> COMMA <y_range> RP
<x_range>              ::=  <integer> | <integer> DD <integer>
<y_range>              ::=  <integer> | <integer> DD <integer>
```

Icon Type

```
<icon_type_def>        ::=  ICON_TYPE <identifier> <icon_def_body> SEMI

<icon_def_body>        ::=  <icon_spec> <icon_components>

<icon_spec>            ::=  ≺       [GENERIC]
                                |   [IS_A <identifier>]
```

```
                          ≻

<icon_components>    ::=  ≺       [<properties_clause>]
                              |   [<pictures_clause>]
                              |   [<labels_clause>]
                              |   [<handles_clause>]
                          ≻
```

Node Component

```
<nodes_clause>       ::=  NODES LP [ <node_spec> {COMMA <node_spec>} ] RP

<node_spec>          ::=  <identifier> AT <location> [<properties_clause>]
```

Link Component

```
<links_clause>       ::=  LINKS LP [ <link_spec> {COMMA <link_spec>} ] RP

<link_spec>          ::=  <link_location> [<link_components>]

<link_location>      ::=  ≺       FROM <location>
                              |   TO <location>
                          ≻

<link_components>    ::=  ≺       [<pictures_clause>]
                              |   [<labels_clause>]
                              |   [<properties_clause>]
                          ≻
```

Edge Type

```
<edge_type_def>      ::=  EDGE_TYPE <identifier> <edge_def_body> SEMI

<edge_def_body>      ::=  <edge_spec> <edge_components>

<edge_spec>          ::=  ≺       [GENERIC]
                              |   [IS_A <identifier>]
                          ≻
```

```
<edge_components>    ::=  ≺      [<properties_clause>]
                            |   [<nodes_clause>]
                            |   [<links_clause>]
                          ≻
```

Appendix B

An Example of EDL/GE Definitions — Data Flow Diagramming

This appendix presents the environment definitions of the Data Flow Diagraming method in EDL/GE language. The definitions are composed of two parts — *Conceptual Definitions* and *Graphical Definitions*. The former defines the *model concepts* of the method, and the latter models their associated graphical representations.

B.1 EDL/GE Definitions of DFD

```
ENVIRONMENT_TITLE "Data Flow Diagrams";

/*****
******   Conceptual Definitions
*****/

VALUE_TYPE
  times_per_unit =
    RECORD (
      quantity : integer,
      unit : (second, minute, hour, day, week, month, year)
     );

ENTITY_TYPE
```

```
  universal GENERIC
    ATTRIBUTES (description : text);

ENTITY_TYPE
  terminator IS_A universal;

ENTITY_TYPE
  data_store IS_A universal
    ATTRIBUTES (id_number : string,
                form : string);

ENTITY_TYPE
  data_element IS_A universal;

ENTITY_TYPE
  process IS_A universal
    ATTRIBUTES (id_number : string,
                  form : string)
    BECOMES process_explosion
      CONNECTIONS flow;

AGGREGATE_TYPE
  any_level GENERIC
    COMPONENTS *
    ATTRIBUTES (description : text);

AGGREGATE_TYPE
  context_diagram IS_A any_level;

AGGREGATE_TYPE
  process_explosion IS_A any_level;

RELATIONSHIP_TYPE
  flow
    ROLES (source, data, destination)
      PARTICIPANTS
        (process, data_element,
           process | terminator | data_store)
        (terminator | data_store, data_element, process)
    ATTRIBUTES (frequency : times_per_unit);


/*****
```

```
******   Graphical Definitions
*****/

CONSTANT FontH  =  13;   /* the default font height */
CONSTANT ProcW  =  80;   /* process icon width */
CONSTANT ProcH  =  8*FontH+2;   /* process icon height */
CONSTANT ProcR  =  18;   /* the radius of the corner rounding
                            for the process picture */

PICTURE_TYPE process_pic
  BOX FROM (0, 0) TO (ProcW, ProcH) RADIUS ProcR
  LINE FROM (0, 2*FontH+1) TO (ProcW, 2*FontH+1);

ICON_TYPE process
  LABELS (id_number AT (ProcW/2, FontH+1)
            PROPERTIES (x_size = 55, y_size = 2*FontH),
          name AT (ProcW/2, 4*FontH+2)
            PROPERTIES (x_size = ProcW-1, y_size = 4*FontH),
          form AT (ProcW/2, 7*FontH+2)
            PROPERTIES (x_size = ProcW-1, y_size = 2*FontH))
  PICTURES (process_pic)
  PROPERTIES (x_size = ProcW+1, y_size = ProcH+1)
  HANDLES (flow.*
           AT ((ProcR .. ProcW-ProcR, ProcR .. ProcH-ProcR)));

CONSTANT DStorW  =  120;   /* data store icon width */
CONSTANT DStorH  =  3*FontH+2;  /* data store icon height */
CONSTANT DStorP  =  31;    /* position of the vertical
                              bar within data store icon */

PICTURE_TYPE data_store_pic
  LINE FROM (0, 0) TO (0, DStorH)
  LINE FROM (0, DStorH) TO (DStorW, DStorH)
  LINE FROM (0, 0) TO (DStorW, 0)
  LINE FROM (DStorP, 0) TO (DStorP, DStorH)
  TEXT "Form: " AT (DStorP+2, 2*FontH+1);

ICON_TYPE data_store
  PROPERTIES (x_size = DStorW+1, y_size = DStorH+1)
  LABELS (id_number AT (DStorP/2, DStorH/2)
            PROPERTIES (x_size = DStorP-2,
                        y_size = 3*FontH),
          name AT (DStorP+1+(DStorW-DStorP)/2, FontH+1)
```

```
             PROPERTIES (x_size = DStorW-DStorP,
                         y_size = 2*FontH),
           form AT (91, 5*FontH/2+1)
             PROPERTIES (x_size = 89, y_size = FontH))
  PICTURES (data_store_pic);

CONSTANT DataW  =  70;       /* data element icon width */
CONSTANT DataH  =  2*FontH;  /* data element icon height */

ICON_TYPE data_element
  LABELS (name AT (DataW/2, FontH)
            PROPERTIES (x_size = DataW-1, y_size = 2*FontH))
  PROPERTIES (x_size = DataW+1, y_size = DataH+1);

/* square size for terminator picture */
CONSTANT TermP  =  6*FontH+2;
/* offset size for the shadow square */
CONSTANT TermD  =  4;
/* the total size of icon side */
CONSTANT TermS  =  TermP+TermD;

PICTURE_TYPE terminator_pic
  BOX FROM (0, 0) TO (TermP, TermP)
  LINE FROM (TermP, TermD) TO (TermS, TermD)
  LINE FROM (TermS, TermD) TO (TermS, TermS)
  LINE FROM (TermS, TermS) TO (TermD, TermS)
  LINE FROM (TermD, TermS) TO (TermD, TermP);

ICON_TYPE terminator
  LABELS (name AT (TermP/2, TermP/2)
            PROPERTIES (x_size = TermP-1, y_size = 6*FontH))
  PROPERTIES (x_size = TermS+1, y_size = TermS+1);

PICTURE_TYPE arrowhead
  LINE FROM (0, 0) TO (-15, -5)
  LINE FROM (0, 0) TO (-15, 5);

EDGE_TYPE flow
  NODES (source      AT (0, 0),
         data        AT (100, 20),
         destination AT (200, 0))
  LINKS (FROM source TO destination
           LABELS (frequency AT (100, -20))
```

```
                PICTURES (arrowhead AT destination));

DIAGRAM_TYPE any_level
  PROPERTIES (x_size = 595, y_size = 770);
```

Appendix C

System Walk-Through of GE Definer

In this appendix, we presents a system "walk-through" of GE Definer using a real-case scenario of defining the "*process*" icon type of the Data Flow Diagraming (DFD) method as an example to illustrate various features and functionality of the tool. There are two reasons why we chose this example scenario to describe GE Definer. Since DFD method is a very popular and well-known graphical SDM, we assume that most readers are already familiar with it and thus able to follow the "walk-through". The second reason is that the selected scenario can show several different features of GE Definer because the "process" icon type is a relatively complex object type.

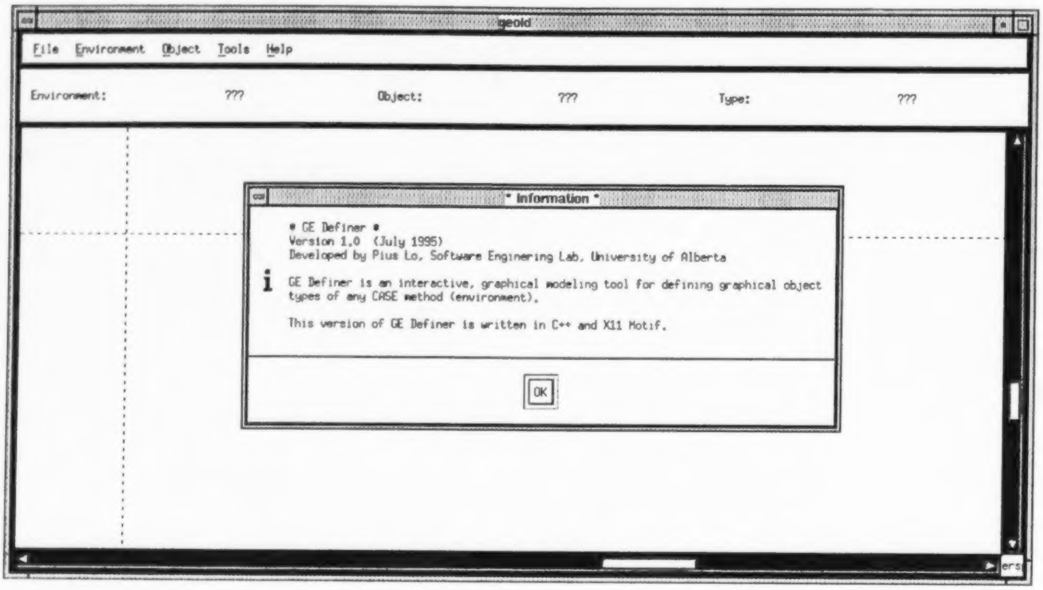
C.1 A scenario of using GE Definer

The example scenario assumes that an existing environment has been created with the definitions of several picture types.

When the method definer[1] starts up GE Definer, the main window, consisting of a menu bar, a status bar, and a canvas area, appears on the screen. The canvas has two scrollbars that can be used to scroll the canvas in two dimensions. There are two dashed lines on the canvas to represent the x and y axes of the drawing area. The user can select an command from the HELP menu to display the information about the current version of GE Definer, as shown in Figure 18.

---

[1] Throughout this section, we use the terms "method definer" and "user" interchangeably to refer to the user of GE Definer involved in this example scenario.



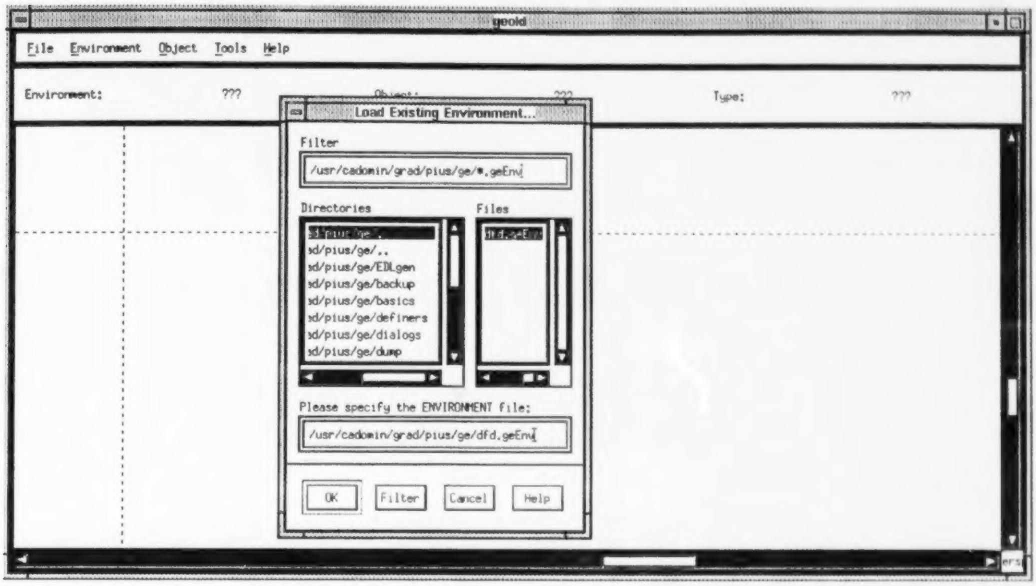
```
Environment:          ???          Object:          ???          Type:          ???
```

```
* GE Definer *
Version 1.0  (July 1995)
Developed by Pius Lo, Software Enginering Lab, University of Alberta

GE Definer is an interactive, graphical modeling tool for defining graphical object
types of any CASE method (environment).

This version of GE Definer is written in C++ and X11 Motif.
```

Figure 18: Starting GE Definer

Because there is already an existing environment defined with picture patterns' specifications, the method definer first loads this environment to GE Definer. S/he selects the `Open` command from the `Environment` menu, and a file selection dialog pops up as shown in Figure 19. The user can therefore select the *Environment File* by either clicking on the directory and file selection windows or entering directly the filename. When s/he specifies the filename and clicks on the `OK` button, GE Definer invokes the GE Parser module to read the file and stores the definitions of the environment in an environment object which is a data structure defined in the Symbol Table. At this moment, this environment object becomes the *current* environment in GE Definer. Its title is displayed on the status bar.

The method definer can now start defining the new object type, "process", by selecting the command `New` under the cascading `Object` menu. The command pops up another



Environment: ???
Object: ???
Type: ???

Filter
/usr/cadomin/grad/pius/ge/*.geEnv

Directories
sd/pius/ge/..
sd/pius/ge/EDLgen
sd/pius/ge/backup
sd/pius/ge/basics
sd/pius/ge/definers
sd/pius/ge/dialogs
sd/pius/ge/dump

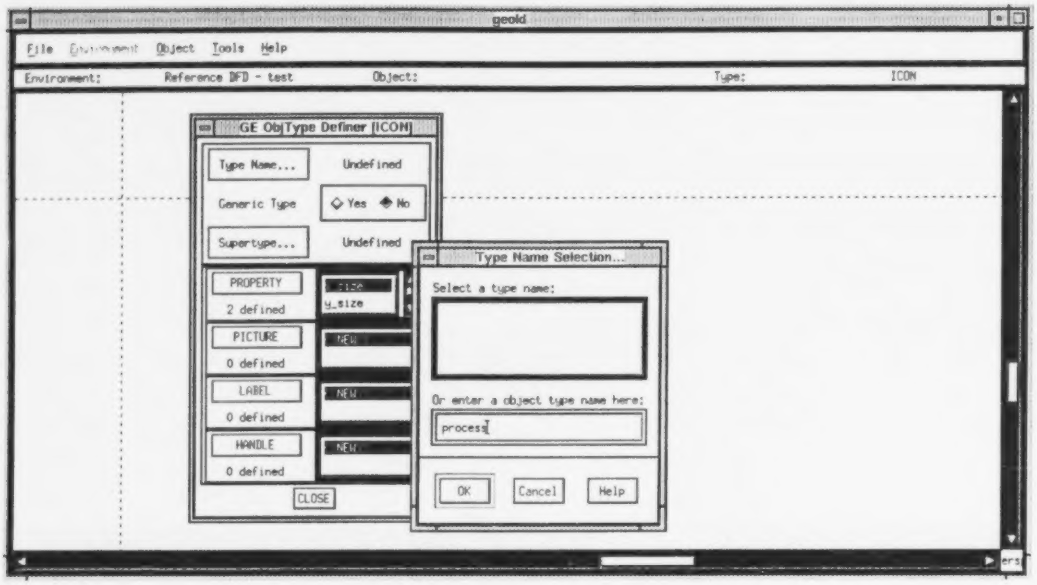
Files

Please specify the ENVIRONMENT file:
/usr/cadomin/grad/pius/ge/dfd.geEnv

Figure 19: Loading *Environment File* in GE Definer

menu with a list of the three GE types: `DIAGRAM Type`, `ICON Type`, and `EDGE Type`. Because "process" is an icon type (which represents the corresponding *entity* type in DFD), the user chooses the `ICON Type` command. At this time, a pop-up *toolbox* appears on the screen. This toolbox is the "GE Object Type Definer", and in this case, it is customized for defining icon type (called "Icon Type Definer"). Since this toolbox is implemented as a dialog window, the user can move it around the screen as s/he wants at any time during the definition process. Figure 20 shows this "Icon Type Definer".

The definer contains a number of buttons and controls to define various properties and components of this new icon type. The user can define the name of this GE type by clicking on the `Type Name` button. A dialog box pops up and the user can input the name through the dialog. This dialog box also provides a selection window where a list of valid type names are shown. These valid choices are collected from the data structures that hold the conceptual definitions of the current environment.



Environment: Reference DFD - test

Object:

Type: ICON

Type Name... Undefined

Generic Type ◇ Yes ◆ No

Supertype... Undefined

PROPERTY 2 defined

PICTURE 0 defined

LABEL 0 defined

HANDLE 0 defined

Or enter a object type name here: process

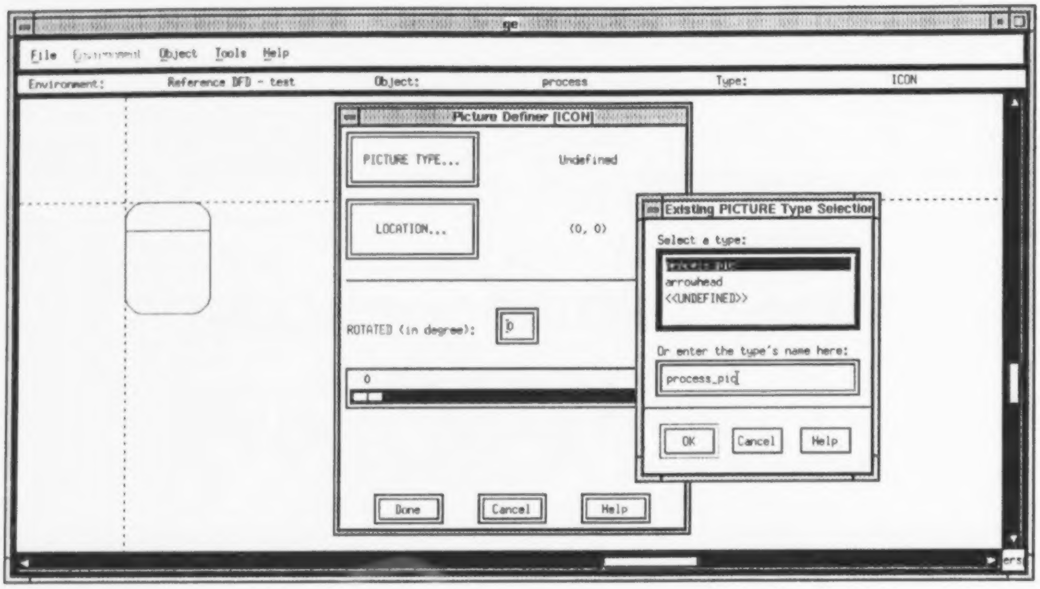
Figure 20: Creating new icon type in GE Definer

Similarly, the method definer can define the name of the supertype by clicking on the corresponding button and enter the name through the pop-up dialog. The supertype is the *parent* of the current object type, and all its components and properties are inherited by the current type. However, because the "process" icon type does not have a parent, the supertype is *undefined*. Furthermore, the user can click on the "radio" buttons to define if this icon type is *generic* or not. If an object type is defined as *generic*, it can be used as a supertype of another object type. Because the "process" icon type is not used as a supertype of any other icon types, and any object type by default is *non-generic*, the user does not need to do anything with these radio buttons. At this point, the user has already defined the standard information for the icon type. S/he can now proceed to define the various components and properties for this object type.

The method definer may first want to attach a picture pattern to this icon type. On the toolbox, there is a selection window next to the `PICTURE` button. This window is used to show all the picture components that are already defined for this GE type. To modify an existing picture component, the user can click on any one of the components' names and then press the `PICTURE` button. On the other hand, to define a new component, s/he can click on a special entry `<<NEW>>` on the selection window. The same procedure also applies to other component types. In this example scenario, because there are no picture components defined yet for the "process" type, the selection window shows only an entry `<<NEW>>`. The message printed right below the button shows the current count of the components defined.

When the user clicks on the `PICTURE` button, the "Picture Definer" toolbox pop ups (see Figure 21). S/he first specifies which picture pattern to be used by clicking on the `PICTURE TYPE` button. A selection dialog appears, which shows the names of the predefined picture types. The user can select the desired picture type from the list. It is important to note that the selection of choices provided by GE Definer is used only as *suggestions* and is never mandatory. This is because, in some cases, the user may want to specify a choice that is not yet defined in the current environment. For example, the user may want to specify a new picture type as a picture component of the current icon type before that picture type is defined. GE Definer is flexible in allowing the definition of *incomplete* object types.

When the user has specified a picture type, the picture pattern is displayed on the screen at the default location — the origin of the icon type. The location can be changed by pressing the `Location` button and specifying a new location on the canvas area. Moreover, the toolbox provides a slidebar as well as a text input box for the user to rotate the picture pattern within the icon type. In our example, since the picture used in the "process" icon type does not requires changes of its default location (i.e., the origin of the picture pattern locates at the origin of the icon type) nor its default orientation (i.e., the pattern is rotated by *zero* degrees). At this point, since the user has completed the definition of the picture component, s/he can click on the `Done` button to close the "Picture Definer". GE Definer then validates the definitions of this picture component for any invalid or missing information. Because the picture component is completely and correctly defined, no warning messages is


ge
File
Environment
Object
Tools
Help
Picture Definer [ICON]
PICTURE TYPE...
LOCATION...
Existing PICTURE Type Selection
arrowhead
<<UNDEFINED>>
OK
Cancel
Help
Done
Cancel
Help

Environment: Reference DFD - test
Object: process
Type: ICON

PICTURE TYPE... Undefined
LOCATION... (0, 0)
ROTATED (in degree): 0

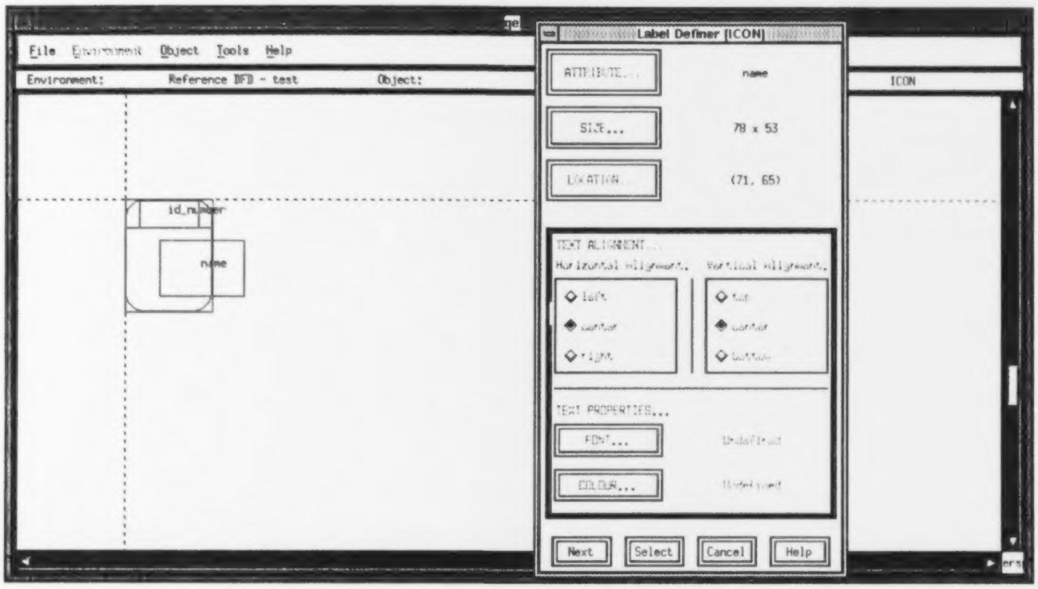
Select a type:
Or enter the type's name here: process_pic

Figure 21: Defining picture component in GE Definer

displayed. GE Definer closes this "Picture Definer" and brings up the "Icon Type Definer" again.

The next component that the method definer needs to define is the label component. Three labels are needed for the "process" icon type in order to show the values of the attributes `id_number`, `name`, and `form`, which are parts of the conceptual definitions of the corresponding entity type. To define a label component, the user basically follows a similar procedure as described above on defining a picture component.

In order to illustrate the *browsing* feature offered by the "definer" toolbox, we consider a situation that the method definer has defined the label components but the `name` label was placed at an inappropriate location. In this case, the user selects an entry of the defined labels from the selection window next to the `Label` button and clicks on the button. GE Definer then pops up the "Label Definer" in its *browsing* mode. During this mode, the toolbox displays the specifications of each defined label one by one, but disables all its controls in order to prevent the user from changing accidentally that label's definitions. The method definer can click on the `Next` button to see the definitions of another label, or click on the `Select` button to switch the "definer" to its *defining* mode and modify the definitions of that label component. Figure 22 shows the "Label Definer" in browsing mode.


File
Environment
Object
Tools
Help
Label Definer [ICON]
id_number
name
TEXT ALIGNMENT...
Horizontal alignment:
Vertical alignment:
Next
Select
Cancel
Help

Environment: Reference DFD - test
Object: ICON

ATTRIBUTE... name
SIZE... 78 x 53
LOCATION... (71, 65)

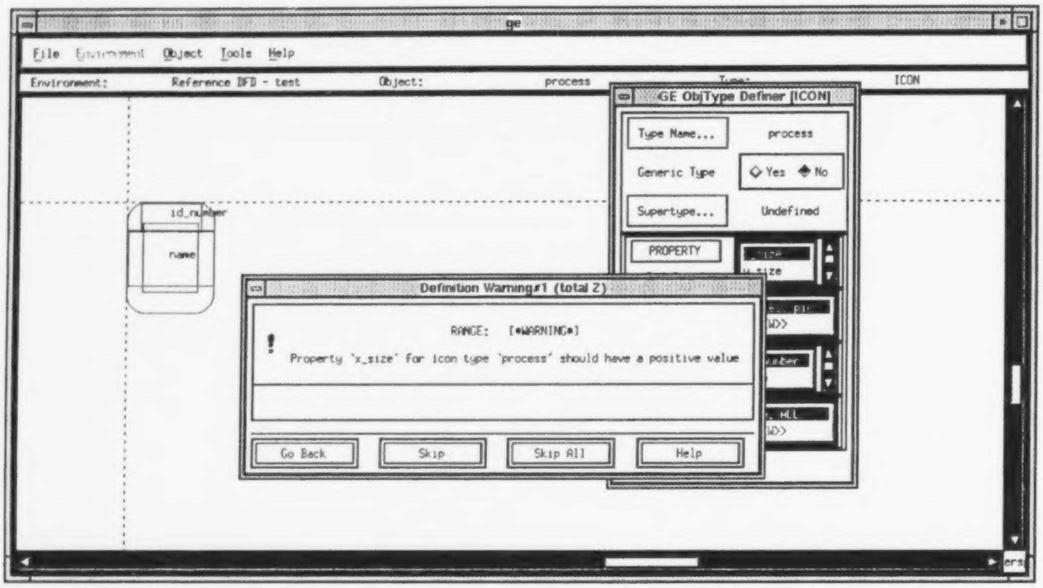
◇ left
◆ center
◇ right

◇ top
◆ center
◇ bottom

TEXT PROPERTIES...
FONT... Undefined
COLOUR... Undefined

Figure 22: Browsing the label component in GE Definer

Finally, the handle component for the "process" icon type is also defined by the method definer using a similar approach. In order to demonstrate how GE Definer employs warning messages, we consider a scenario that the method definer has defined everything *but* the size of the "process" icon type, and s/he does not notice that this part is missing. When the method definer closes the "Icon Type Definer", GE Definer validates the definitions of the icon type and detects this missing property. Hence a warning dialog is displayed on the screen, as shown in Figure 23. The warning dialog reports that there are a total of two warnings generated. It is because an object type's size is defined by two independent properties — `x_size` and `y_size`. The warning window displays each warning, one by one, with its *kind*, *severity*, and *explanation*. The user can then react in one of the three possible ways: *go back and fix the problem*, *read the next warning message*, or *skip all the warnings*. In order to define properly the "process" icon type, the method definer should go back to the "definer" and define the size of the type.



Environment: Reference DFD - test
Object: process
Type: ICON

Type Name... process
Generic Type ◇ Yes ◆ No
Supertype... Undefined

RANGE: [*WARNING*]

Property 'x_size' for icon type 'process' should have a positive value

Figure 23: Warning display in GE Definer

After all the graphical components and properties are properly defined, the definition of the "process" icon type is complete. To store this icon type in the current environment, the method definer executes the command "`Add to Environment`" under the `Object` menu. Before the icon type can be appended to the environment, GE Definer must validate this type again *with respect to* the environment. This validation process is different from the one done by the "GE Type Definer" because it ensures not only that the definitions of the object type itself are complete and valid, but also that the definitions are *consistent* with those of other object types in the environment. In other words, an environment does not accept any new object type that causes conflicts with the existing definition. Once GE Definer has successfully validated and attached the "process" icon type to the current DFD environment, the method definer can execute the `Save` command in the `Environment` menu and save the environment, in the form of EDL/GE definitions, back into the original *Environment File* (or a new file). The output of this Environment File is presented in Section C.2. We recommend that the interested reader to compare the output generated by GE Definer with the definitions produced manually as shown in Appendix B. The tool-generated output uses indentations, proper line breaks, and pre-defined comments in order to make the definitions as readable as those produced manually. However, a noticeable difference between the definitions produced in these two ways is that the output generated by the GE Definer does not use symbolic constants. This is because the current version of GE Definer is not capable of defining new constant values. However, the tool is able to *preserve* the constant values used in an object type if these constants are previously defined in the Environment File. The capability of defining symbolic constants will be one of the future extensions to GE Definer.

In summary, GE Definer allows the method definer to build the graphical environment of a method in an *incremental* way. That is, s/he does not have to define the whole environment at once; instead, new graphical object types can be defined and added to the environment at any time in order to make the environment more complete.

C.2 Environment File Generated by GE Definer

```
ENVIRONMENT_TITLE "Reference DFD - test";

/*
This environment is generated by GE Definer.
*/

//======================//
// GRAPHICAL DEFINITIONS START HERE:
//----------------------//
PICTURE_TYPE process_pic
     BOX FROM (0, 0) TO (80, 106) RADIUS 18
     LINE FROM (0, 27) TO (80, 27)
;

PICTURE_TYPE arrowhead
     LINE FROM (-15, -5) TO (0, 0)
     LINE FROM (-15, 5) TO (0, 0)
;


//----------------------//
ICON_TYPE process
     PROPERTIES ( x_size=81, y_size=107 )
     PICTURES ( process_pic AT (0, 0) ROTATED 0 )
     LABELS ( id_number AT (40, 14)
              PROPERTIES ( x_size=55, y_size=26 ),
              name AT (40, 54)
              PROPERTIES ( x_size=78, y_size=53 ),
              form AT (40, 93)
              PROPERTIES ( x_size=78, y_size=26 ) )
     HANDLES ( flow.* AT ( (15..65,20..86) ) )
;
```